AD-A056 110 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
ARCHITECTURE FOR MULTIPLE INSTRUCTION STREAM LSI PROCESSORS.(U)
OCT 77 W J KAMINSKY DAAB07-72-C-0259
UNCLASSIFIED R-796 NL

1 OF 2
ADA
056110

REPORT R-796 OCTOBER, 1977 (12) UILU-ENG 77-2243

COORDINATED SCIENCE LABORATORY

ARCHITECTURE FOR MULTIPLE INSTRUCTION STREAM LSI PROCESSORS

WILLIAM JOSEPH KAMINSKY, JR.

UNIVERSITY OF ILLINOIS - URBANA, ILLINOIS

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) ARCHITECTURE FOR MULTIPLE INSTRUCTION STREAM LSI PROCESSORS.		5. TYPE OF REPORT & PERIOD COVERED (9) Technical Report.
		6. PERFORMING ORG. REPORT NUMBER (14) R-796, UILU-ENG-77-2243
7. AUTHOR(s) (10) WILLIAM JOSEPH/KAMINSKY, JR.		8. CONTRACT OR GRANT NUMBER(s) (15) DAAB-07-72-C-0259, NSF-MCS 73-03488 A01
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		12. REPORT DATE (11) October, 1977
		13. NUMBER OF PAGES 120 (12) 131p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Pipelining
Precedence Graph
Multiprocess

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Advancements of the semiconductor industry have produced an exponential increase in the capacity of integrated circuit chips. As this trend continues, much more complex systems will be able to be implemented on a single chip. This research is aimed at studying computer architectures which are most suited for LSI implementation. A pipelined multiprocessor architecture is derived and methodology given which determines a layout for such a processor from a given instruction set. The problem of partitioning a processor when it is too large for one chip is also examined. An evaluation of the

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE 78 07 10 160 UNCLASSIFIED

097700

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

CL

20. ABSTRACT (continued)

performance of the pipelined processor as compared to a single stream processor is made. Three program traces are analyzed to study their usage of registers and this data is used to determine the performance of the processors with various degrees of pipelining and number of data registers.

UILU-ENG 77-2243

ARCHITECTURE FOR MULTIPLE INSTRUCTION STREAM LSI PROCESSORS

by

William Joseph Kaminsky, Jr.

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259 and in part by the National Science Foundation under Grant MCS 73-03488 A01.

ARCHITECTURE FOR MULTIPLE INSTRUCTION STREAM LSI PROCESSORS

BY

WILLIAM JOSEPH KAMINSKY, JR.

B.S., Purdue University, 1973
M.S., University of Illinois, 1974

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Thesis Adviser: Professor Edward S. Davidson

Urbana, Illinois

ARCHITECTURE FOR MULTIPLE INSTRUCTION STREAM LSI PROCESSORS

William Joseph Kaminsky, Jr., Ph.D.
Department of Electrical Engineering
University of Illinois at Urbana-Champaign, 1977

Advancements of the semiconductor industry have produced an exponential increase in the capacity of integrated circuit chips. As this trend continues, much more complex systems will be able to be implemented on a single chip. This research is aimed at studying computer architectures which are most suited for LSI implementation. A pipelined multiprocessor architecture is derived and methodology given which determines a layout for such a processor from a given instruction set. The problem of partitioning a processor when it is too large for one chip is also examined. An evaluation of the performance of the pipelined processor as compared to a single stream processor is made. Three program traces are analyzed to study their usage of registers and this data is used to determine the performance of the processors with various degrees of pipelining and number of data registers.

ACKNOWLEDGMENT

The author wishes to extend his deepest gratitude to his advisor. Professor Edward S. Davidson. Dr. Davidson's perceptive guidance and generous patience as well as his careful review of this manuscript were indispensable to this research.

The author also wishes to thank his colleagues for their contributions to this research, namely, Balasubramanian Kumar and Dan Hammerstrom for their programming help along with Rob Budzinski, Joel Emer and Alan Grant for their constructive comments and discussions with respect to this research.

The typing of Mary McMillen and the support of the CSL staff are also acknowledged.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

Page

CHAPTER 1. INTRODUCTION

1.1. Current State of the Art

Advancements made by the semiconductor industry have greatly increased the complexity of a single integrated circuit chip. Over the nearly 20 year history of the industry there has been an exponential increase in the complexity of integrated circuits [NOY76] and it is expected that this trend will continue. Accompanied by this increase in chip complexity is a reduction in cost per gate of digital systems implemented with LSI. One of the most dramatic recent achievements of the industry is the implementation of small CPU's, called microprocessors. As this trend of increasing chip size and density continues, it will enable larger processors with greater capabilities to be implemented with a single chip. Not only will processors become more complex, but semiconductor memories are also getting faster, larger and cheaper. Entire computing systems will become smaller, more powerful and cheaper, enabling the use of more powerful computers in applications where it is now infeasible. In order to produce efficient computers with large scale integrated circuits, the characteristics of large scale integration must be studied. The purpose of this research is to examine the architectures suitable for LSI implementations of future processors.

The objective of future LSI processor design is to use the increasing chip area available to increase performance as much as possible. To obtain maximum performance from future LSI processors, the factors affecting cost of LSI processors must be studied and suitable designs determined. Many present microprocessors were adapted for LSI from existing discrete computer designs. These discrete designs in turn were originally designed to reduce gate count and based their processor cycles upon core memory cycles. Relevant factors such as pin count and interconnection patterns were only considered secondarily. Future LSI designs must be based on other criteria.

The two main cost factors for LSI implementation are pin count and chip area. With exponentially increasing chip area, chip area capacity no longer limits the number of components as seriously as the number of pins allowed for connecting the internal components to the rest of the system. Although chips with more pins are being produced, physical constraints such as board area, practical chip package size and packaging costs do not allow a rapid increase of pins.

Large numbers of pins contribute to other costs as well. Area costs are increased by large numbers of pins since each pin requires a pin driver which uses a significant amount of area due to its need to handle higher currents and voltages than internal devices. Connection pads are needed as well. A greater number of pins increases system costs since the number of PC board connections and wiring interconnections tends to increase with more pins. Reliability is reduced also since many chip failures are due to pin faults [KET76] and many system failures result from broken PC board connections. It is therefore important to reduce the number of pins to obtain a cost effective and reliable system. This goal should be kept in mind from the inception of the design process.

The other cost factor, chip area, primarily affects the yield of the fabrication process. Yields drop precipitously as the area exceeds some practical limit. The practical area limit is increasing, but efficient use of the area allowed is important to obtain maximum cost effectiveness. On-chip interconnections, in many cases, compose as much as 67 percent of the total chip area [PEN72], especially for "random logic" implementations, which comprise a significant portion of microprocessor layout. Reduction of on chip interconnection area will increase the number of active devices possible within a chip. Future designs should strive for regular structure since this leads to less interconnection area and a higher component/cost ratio.

1.2. Future LSI Designs

There are several ways of utilizing additional chip area to increase the system performance of future LSI processors. One way is to increase the instruction repertoire which utilizes additional area for instruction decoding and control signal generation. This reduces the number of instructions needed to execute a program and may affect the average instruction execution time. More complex computational capabilities could be added such as multiply units or floating point units which would reduce the time for execution of numerical operations. The register set could be increased to reduce the number of memory references required and thereby reduce execution time. Multiple processors could be implemented on one chip to process multiple programs at one time. Pipelining could be used to increase throughput of the processor or as a way of achieving multiprocessing with shared hardware instead of using completely separate processors.

The two objectives for LSI designs, few pins and efficient use of chip area, must be considered in choosing which methods of improving processor performance are used. Reducing pins is probably the most important objective. This research is aimed primarily at this objective.

First, consider the pin usage in a typical microprocessor. A typical microprocessor usually has cycles composed of several phases. During one phase an operation or data transfer is accomplished. If there needs to be information flow into or out of the CPU, a pin or set of pins will be used to transfer this information. Usually each pin is dedicated to one signal path. In some cases, pins may be shared by mutually exclusive signals when there is no ambiguity as to which signal is being transferred. Many of these pins, such as memory address or data pins, will be used only once during an entire cycle and remain idle during the rest of the cycle. Idle pin time of a single processor chip cannot be reduced greatly in some cases since more pins are required during some phases than during other phases and utilization of all pins during each phase cannot be accomplished. However, if the pins are utilized well on a chip, the chip is likely to require fewer pins. Let the pin utilization η_p be defined as follows.

$$\eta_p = \frac{\text{average number of phases per cycle during which a pin is used}}{\text{total number of phases per cycle}} . \quad (1.1)$$

Let the average for the chip be $\bar{\eta}_p$ where n = number of pins and

$$\bar{\eta}_p = \frac{1}{n} \sum_n \eta_p . \quad (1.2)$$

For any particular design $\bar{\eta}_p$ should be as close to one as possible.

One way to obtain high performance and high pin utilization is to use several independent processors on one chip as mentioned previously. A chip containing N processors operating independently, but shifted in time so that the pins could be shared using a multiplexer/demultiplexer is shown in Figure 1.1. Here the processors would have identical cycling successively shifted in time from processor to processor by one phase. Furthermore, for each pin, each processor would only have access to the pin during a particular phase of each cycle. Each pin always transfers a particular signal, but for successive processors in successive phases. For this system, the pin

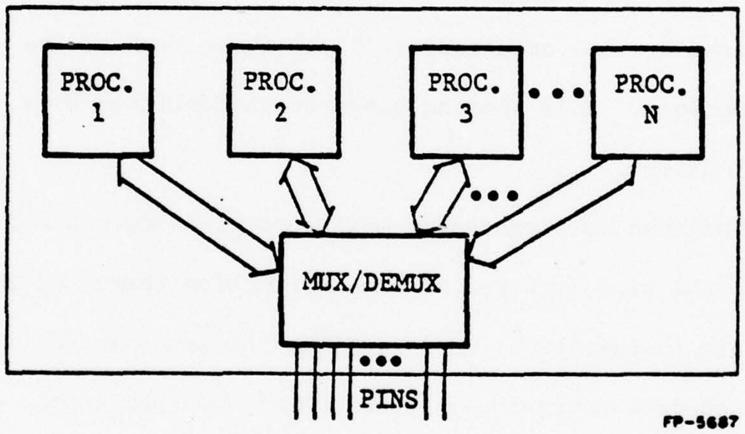


Figure 1.1. Set of Several Independent Processors in One Chip.

efficiency, $\bar{\eta}_p$, will be close to one and the performance will be approximately N times that of one of the processors (provided interference between accesses to memory or I/O devices is not significant and all processors can be kept busy).

There are several problems with this scheme. Processors must be timed synchronously with respect to each other. It would require a very large chip area to hold N complete processors plus the multiplexer/demultiplexer circuitry. This design would also require long interconnections from each processor to the one multiplexer/demultiplexer circuit. One way to reduce the area required by this architecture is to share much of the hardware units and interconnections. This sharing could be accomplished by a pipelined multiprocessor design.

A pipelined multiprocessor which processes multiple instruction streams within the pipe (instead of one instruction stream of which several instructions are in execution) would achieve the same pin utilization as the above set of separate processors. Each segment of such a pipe would execute one phase of one process and control the set of pins associated with that segment of the pipe. The pipelined processor would, instead of having N copies of each hardware unit, only have as many copies as are used during one cycle of execution and share them amongst all processes. Since the pipelined architecture employs a uniform flow of data through the pipe, it requires a highly regular structure to be implemented on a chip, thereby reducing the interconnection area required and providing automatic timing of all processes in the pipe. Since pins are connected directly to one segment of the pipe, no multiplexer/demultiplexer hardware is required. This pipelined architecture combines increased performance, efficient use of chip area and high pin utilization. Since this pipelined approach has many of the characteristics suitable for LSI implementation, this research is aimed at developing this architecture. The pipelined architecture performance is also compared with multiple data register processors since the comparison is straightforward and the multiple data register approach is used in some microprocessors and many minicomputers to improve performance.

1.3. Research Outline

Chapter 2 develops a design methodology for a pipelined LSI multiprocessor introduced in the previous section. This methodology begins by using a given instruction set to define the basic operations of the processor. The precedence of operations for each instruction is determined and used to define a shared set of resources. Timing of these resources is analyzed and merges found to reduce still further the number of interconnections and hardware units. The physical structure is then composed from a merged resource utilization graph.

Chapter 3 studies the pin problem when a partition of the system into several chips is necessary. Three possible methods are described of which two are chosen (a bit slice partition and register partition) for further modeling and comparison of effectiveness. The advantages and disadvantages of each partitioning scheme and a combination of the two schemes are summarized.

Chapter 4 studies the performance of the pipelined processor vs. a multiple data register processor. This performance study is based on a memory port limited processor, since pin considerations limit the chip to one memory port. The register usage of programs run on an IBM 360 is analyzed to determine the dependence of performance on the register set size. This data is then used to compute the performance of the pipelined processors with various degrees of pipelining and numbers of data registers.

CHAPTER 2. DESIGN METHODOLOGY FOR A PIPELINED LSI MULTIPROCESSOR

2.1. Processor Definition

The first step toward obtaining a pipelined organization is to define the machine in terms of the functions it must perform as derived from a given instruction set and register set. The instruction set is divided into two parts, the user instructions, which are those needed by the user to perform his computational requirements (such as add, move, increment, etc.) and system instructions, which are those needed by the system to perform operating system tasks and other controlling operations (such as interrupt handling instructions, I/O control, etc.). The user instruction set should be selected to achieve efficient execution of the types of programs to be run on the processor. These instructions need to assume certain details of the processor, such as addressing methods and the size and nature of the addressible register set, but should be defined in as machine-independent a manner as possible. The system instructions must also be given and are used to carry out additional operations which must be performed by the processor. The architecture is then tailored to best execute these two sets of instructions. It is also helpful to be given estimated instruction frequencies for the set of instructions, since it is important to optimize execution of highly used instructions.

Along with the instruction set, a set of registers is given which are explicitly referenced by the instructions (data registers, index registers, etc.). Next an implicit set of registers which are required by the processor to execute the instruction set and to perform any operations are derived (memory address and data registers, program counter, etc.). These two sets of registers must be implemented in the processor either as physical registers actually contained in the CPU, or as virtual registers which are kept in memory, but easily accessed by the CPU.

The instruction set and the set of registers defined above are used to determine the physical structure of the pipelined CPU. This structure consists of functional units, registers and communication paths. The following sections describe the steps involved to realize the final layout for LSI implementation.

To aid the explanation of the procedures, an example will be carried through this chapter illustrating the steps required. The example is for illustrative purposes only and is not intended as a general result to be actually implemented in practice.

The example chosen utilizes a subset of the instruction set of the HP-2100 minicomputer [HP]. Table 2.1 lists the instructions used in the example. The instructions for I/O control have been omitted since these are specifically for the I/O system of the HP computer. The I/O control would be implemented in the control section of the pipelined processor, which is beyond the scope of this research. Only the processing section is investigated in this research.

The register set chosen to be used in the example consists of a Memory Address Register (MA), a Memory Data Register (MD), a Program Counter (PC), an Instruction Register (IR), a Stack Pointer (SP), and an Accumulator (A). Only one accumulator was chosen since a mimimal register set is desired and the instruction set does not require any specific quantity of registers (although the actual HP-2100 implementation has two, called the A and B registers). The SP register has been added to facilitate the use of recursive procedures.

Table 2.1. Instructions Used in Example

CLR	CLEAR A
CLR $\begin{pmatrix}N\\Z\\C\end{pmatrix}$	CLEAR STATUS BIT N, Z OR C
SET $\begin{pmatrix}N\\Z\\C\end{pmatrix}$	SET STATUS BIT N, Z OR C
DEC	DECREMENT A
INC	INCREMENT A
COMP	COMPLEMENT A
NEG	NEGATE A
TEST	SET STATUS BITS ACCORDING TO CONTENTS OF A
ASL	ARITHMETIC SHIFT A LEFT
ASR	ARITHMETIC SHIFT A RIGHT
ROL	ROTATE A LEFT
ROR	ROTATE A RIGHT
LOAD	LOAD A FROM MEMORY
STORE	STORE A TO MEMORY
ADD	ADD MEMORY LOCATION TO A (RESULT IN A)
SUB	SUB MEMORY LOCATION FROM A (RESULT IN A)
XOR	EXCLUSIVE OR A AND MEMORY LOCATION (RESULT IN A)
IOR	INCLUSIVE OR A AND MEMORY LOCATION (RESULT IN A)
AND	AND A AND MEMORY LOCATION (RESULT IN A)
JMP $\begin{pmatrix}N\\Z\\C\end{pmatrix}$	CONDITIONAL JUMP IF N, Z OR C TRUE
JMP	UNCONDITIONAL JUMP
JSUB	JUMP TO SUBROUTINE
RSUB	RETURN FROM SUBROUTINE
WRIO	WRITE TO I/O DEVICE
RDIO	READ FROM I/O DEVICE

2.2. Basic Assumptions

Several basic assumptions were made before proceeding to determine the physical structure of the CPU. The first assumption deals with the control section of the processor. The control section can be treated separately from the processing section (that section which performs operations on and transfers data) and is linked to the processing section by the necessary control and sense lines. It is assumed that a control section exists which receives the instruction and necessary sense signals and generates the control signals for the processing section, external memory and I/O devices. Methods of implementation of control applicable to this type of processor have been described by Kogge [KOG77] and will not be pursued here.

The second assumption is that the processor will contain only one Memory Address port and one Memory Data port. If two or more ports were used, the number of memory pins would be multiplied by the number of ports and it would be difficult to obtain full utilization of these pins. A multiple port memory also increases the complexity and cost of the memory and busses which is likely not to be cost-effective.

The third assumption is that I/O devices are treated as memory locations and communications to them are made over the memory data and address busses in a similar fashion to that used by the PDP-11 [DEC74]. This assumption also reduces pins since if a separate port is used, pins would have to be added to the CPU chip and τ_p would be very small for these pins. Using the memory bus for I/O communications requires the loss of a memory cycle each time an I/O transfer is made (if memory otherwise would be requested during the same cycle as an I/O information transfer), but this should not produce a significant degradation of performance since I/O devices run much slower than the CPU. Any control signals generated by the CPU control section and pins for I/O control (e.g. interrupt, flags, acknowledgment, etc.) would be added to the CPU control section.

Special purpose hardware devices (e.g. floating point unit, multiply, etc.) could be connected in the same fashion as I/O devices. These communicate with the CPU over the memory busses. This assumption maintains high pin utilization and minimal pins, although it also eliminates a possible memory reference for each special device transfer cycle.

2.3. Functional Components

Given the instruction set and register set, a method is needed for describing each instruction in terms of which resources (a resource is any communication path, register or functional unit) are required, what operations must be performed and in what order they are performed. To do this each instruction will be decomposed into a set of functional components. These functional components are register-to-register transfers, functional operations and memory accesses.

A register-to-register transfer involves only a read from one register of its contents and a write of this information into another register. This functional component involves the use of two registers and a connection path between them.

The second type of component involves a read from one or more registers to a functional unit (a functional unit is a hardware unit which performs a functional operation (add, sub, exclusive or shift, clear, etc.)) which performs the necessary operation. The result is then written to the result register. Each functional unit is specified by the types of operations it actually performs and not given a general name such ALU. This specification defines exactly the type of device needed (this minimizes the complexity of the functional unit for each unit required).

This minimal complexity description of each functional unit enables distribution of appropriate functional units throughout the pipeline. This is useful for two reasons. Multiple units allow operations to be performed in parallel, thereby speeding computation. Secondly, distribution of the functional units reduces multiple communication paths to one unit, thereby reducing the required interconnection area. In fact, in some cases, the added functional units may require less area than the interconnections would require with fewer units. The resources needed for each functional component of the second type are the input data register(s), the communication path(s) from the input register(s) to the functional unit, the functional unit, the communication path from the functional unit to result register and the result register.

The third functional component, the memory access, first consists of a transfer of the address from the MA register to the memory address bus. Then, at the appropriate time, a write of the data from the memory data bus to the MD register for a read operation or a write to the memory data bus from the MD register for a write operation. When describing this functional component the memory address and data busses are treated as registers.

2.4. Instruction Precedence Graph

To describe each instruction, a precedence graph is formed which indicates which functional components are needed and the partial order in which they must be performed for that particular instruction. The precedence graph is composed of a node representing each functional component and directed arcs joining these nodes indicating the precedence of execution. Any combination of functional components which may be performed in parallel is indicated by parallel paths in the graph.

As an example, Figure 2.1 shows the precedence graph and the definition of each node in the graph for the ADD instruction. This instruction adds the contents of register A and the contents of the memory address specified in the instruction and places the result in register A. By examining the graph in Figure 2.1, one can see that nodes 1,2 and 3 and node 6 are in parallel paths and can be executed in parallel. The same occurs with nodes 4 and 5 and nodes 7 and 8. However, node 7 requires both nodes 3 and 6 to be performed before it but node 4 only requires node 3 to be performed before it, as indicated by the directed arcs. Once nodes 5 and 8 have been completed, this instruction is finished and the next instruction may begin.

A precedence graph for each instruction must be formed before proceeding to the next step. Appendix A contains the precedence graphs for all the HP-2100 instructions shown in Table 2.1.

The precedence graph for each instruction begins when data comes from memory via the data bus and is transferred from the memory data bus to the IR and MD. In fact, this is also used as the beginning of the first processor cycle for each instruction. The cycle time for the pipelined processor is the time it takes a process to pass through the entire pipeline and return to the beginning of the pipe.

Since the processor has only one memory port, as described in Chapter 1, only one memory reference can be made by a process during one processor cycle. Although instructions are commonly thought of as starting with the fetch (after PC → MA) no execution of the fetched instruction can be

Add : Reg A + (MEM Address) to Reg A

1) $MEM_{DATA} \longrightarrow IR, MD$

2) $MD_{ADDRESS} \longrightarrow MA$

3) $MA \longrightarrow MEM_{ADDRESS}$

4) $MEM_{DATA} \longrightarrow MD$

5) $MD + A \longrightarrow A$

6) $PC + 1 \longrightarrow PC$

7) $PC \longrightarrow MA$

8) $MA \longrightarrow MEM_{ADDRESS}$

$$\begin{tikzcd}
{} \arrow[r] \arrow[dr] & 1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] \arrow[drr] & & 4 \arrow[r] & 5 \arrow[r] & {} \\
& 6 \arrow[rrrr] & & & & 7 \arrow[r] & 8 \arrow[ur] &
\end{tikzcd}$$

FP-5688

Figure 2.1. Functional Components and Instruction Precedence Graph for the "ADD" Instruction.

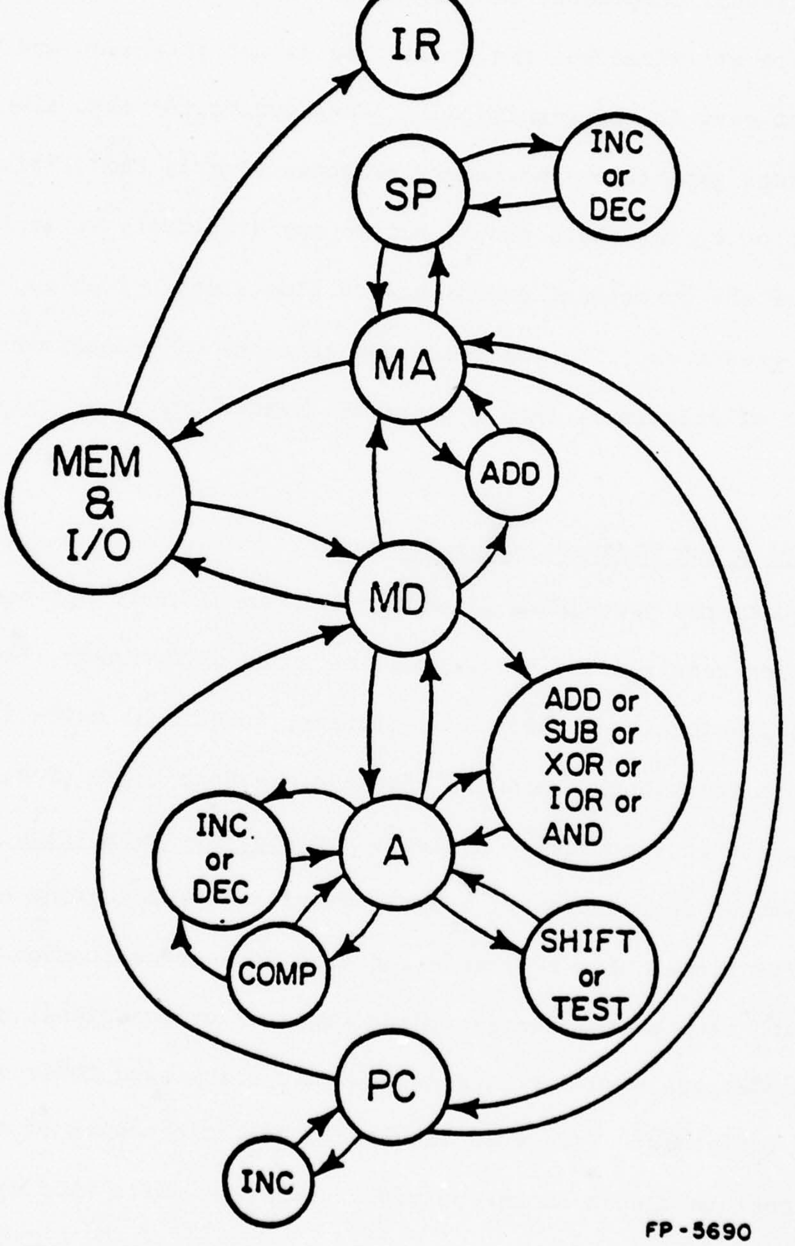
performed prior to receiving the instruction from memory. The completion of the previous instruction could be overlapped with the fetch provided generation of the fetch does not require any resources needed to finish the previous instruction. The memory data bus to IR and MD reference point is chosen only as a convenient point to start each instruction cycle, since overlap across this point is not likely to be possible. However, this choice does not prohibit overlap at this point between instructions.

Since any particular cycle usually begins with the functional component MEM DATA $\rightarrow$ MD, the remaining functional components can be executed at various times within a cycle, possibly some in parallel, dependent upon the precedence relationship, the number of resources and their position in the pipe. The <u>time assignment problem</u> is to determine the position in time at which each of these functional components is performed. This assignment must be made taking into consideration the <u>resource assignment problem</u>, namely, determining which communication paths are required, the types of functional units and the number required of each type. The objective of both of these assignments is to strike a balance between the minimization of execution time and the minimization of resources.

The run time of a program is significantly influenced by the number of memory references when it is run on a pipelined processor with one memory port. The performance can be evaluated by the number of memory references and the time between them. If the architecture is to be changed to increase the performance, either the time between references must be reduced or the number of memory references required must be reduced. When evaluating the time assignment problem, the critical path (the path of the precedence graph which takes the longest time) tends to be the memory address generation and memory response time. The functional components necessary to generate the memory address should be executed as soon as possible in the cycle. The rest of the functional components should be fit into the cycle so that distinct resources are minimized but the cycle time is not increased and instructions for the most part do not require additional cycles for execution. When the memory address generation and memory response time is the critical path for an instruction cycle, there should not be any difficulty in assigning the remainder of the functional components to time slots. Problems arise only in minimizing resources. The following two sections will determine first, the minimal set of resources, then a practical method for time assignment of resources.

2.5. Defining the Minimum Set of Resources

Once the precedence graph for each instruction has been developed, the resources required for the processor can be determined. The resources, as defined previously, consist of registers, functional units and communication paths. Each functional component of the precedence graph of each instruction requires a set of resources. A single register transfer graph can be composed from the set of instruction precedence graphs by representing each register and each functional unit by a node and each data communication path by a directed arc between the corresponding register or functional unit nodes. Separate nodes are used for functional units which send their results to different registers. This rule results in multiple copies of the same type of functional unit such as the multiple increment units used by the PC, SP and A registers in Figure 2.2. Multiple functional units are used to reduce the number of communication paths. Using only one functional unit of each



FP-5690

Figure 2.2. Register Transfer Diagram.

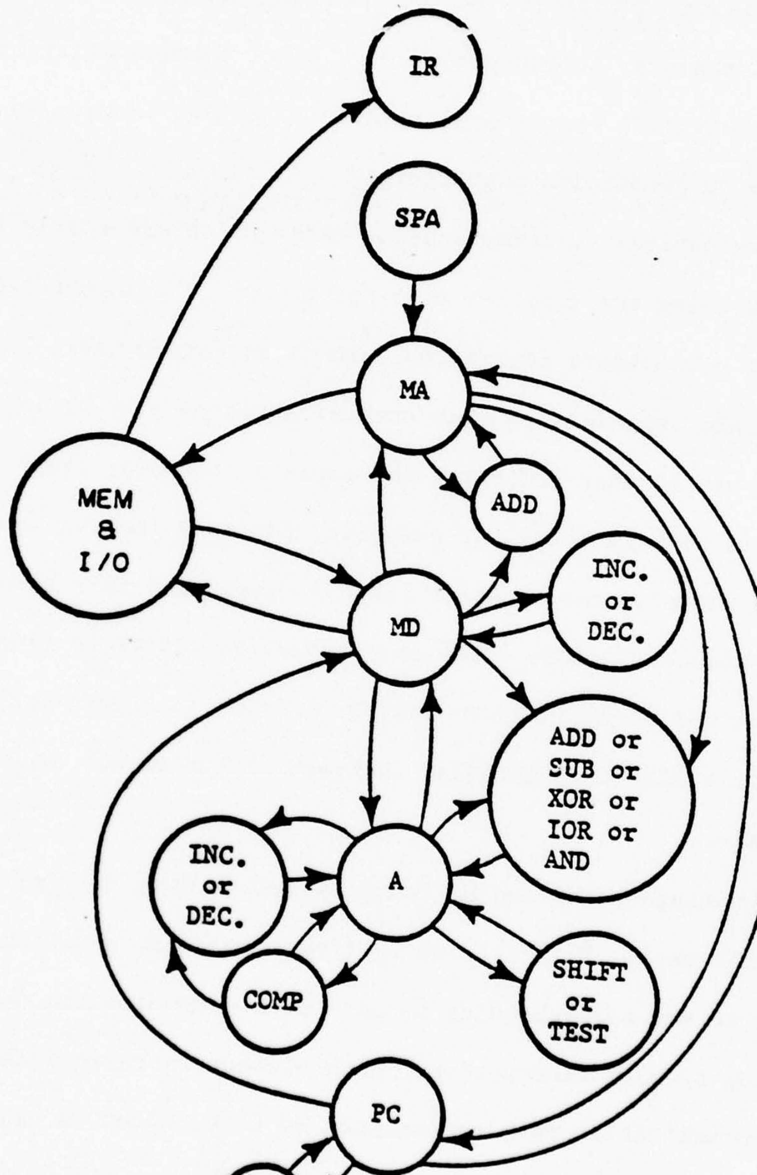
type would require connection paths from each register which uses the functional unit, thereby increasing the interconnection area required. This restriction represents a significant departure from the traditional register-bus-ALU type of organization. The departure is motivated by an appreciation of LSI technology in which replication of simple functional units should take less area than the interconnection area required to share a single copy of each type of functional unit. Also, more than one use of any single functional unit during any one cycle would cause an additional copy of this functional unit anyway unless additional cycles were used to execute the instruction. A register transfer graph for the example is shown in Figure 2.2.

This graph was composed using the following method. First a node is used to represent each register in the processor and one for memory. It is helpful to place together nodes which represent registers which are likely to have many data communication paths between them. I/O devices or channels may also be represented with nodes, but in this example I/O is handled as a subset of memory and is included in the memory node. Secondly, the instruction precedence graphs are examined one by one and for each functional component, the necessary data communication paths and functional units are determined. If a data communication path is not represented in the data transfer graph, an arc is added between the corresponding register nodes or functional unit nodes to represent that data transfer path. If a node does not yet exist to represent a functional unit as required by the functional component, a node is added for this functional unit and arcs are added from the functional unit to the result register and from the input registers to the functional unit. After all instruction precedence graphs have been examined, the register transfer graph is complete.

This graph shows a minimum set of interconnection paths between registers and functional units (at the expense of using extra functional units of some types). Using this graph, changes in the complexity of interconnections can be studied as changes of the architecture or instruction set are made. The graph is further useful to determine effective CPU partitioning into several chips. If a partition were desired, a grouping of registers and functional units which had few interconnection paths between them is required to minimize pins. This graph can be used to determine highly connected registers although a different version of this graph is used in Chapter 3 so that partitioning can be studied more directly.

Changes of architecture and organization can be evaluated with the graph by determining the addition or reduction of data communication paths for each change. Having a register in the processor can be traded off against using a memory location to store its contents and adding a cycle for the memory reference. This is done by changing all instruction precedence graphs to correspond to the change and redrawing the register transfer graph. The effect of the change on interconnections can be observed and a decision can be made to accept it or not.

An example of this tradeoff for the instruction set used is shown in Figure 2.3. Here the stack pointer register is removed and its contents are stored in a fixed location of memory. This change results in the elimination of a register, but adds two communication paths between the MD register and the MA register. Also, the functional unit performing additions, subtractions and logical operations now has three inputs instead of two. These results show that removing this register increases connection complexity and would likely be more costly than including the register itself. (A more



FP-5689

Figure 2.3. Modified Register Transfer Diagram.

accurate cost estimation requires data about the technology used for implementation.) A decision, therefore, was made to keep the stack pointer register as a physical register in the processor. Other changes of architecture and processor organization can be evaluated in a similar fashion to determine their effects on connection requirements.

Alternatives to communication paths which are costly to implement can be sought using the register transfer graph. The instructions that use a path which is a candidate for removal are first determined. This is done by labeling the arc representing each communication path according to which instructions use it when composing the register transfer graph. An alternative for using this path is found, if possible, for each instruction which uses the path. A new register transfer graph is drawn for this change and the differences noted. In some cases an alternative cannot be found (such as for the path from the PC register to the MD register in Figure 2.2). This is used only for a JUMP to SUBROUTINE instruction but cannot be eliminated and so must remain.

Low usage paths can be found by determining the probability of usage for each path. This is done by first composing a matrix called R_T, which has a column corresponding to each data communication path and a row corresponding to each instruction. Each element represents the number of times the communication path represented by that column is used by the instruction represented by that row.

$$R_T = \begin{array}{l} \\ \text{ADD} \\ \text{OR} \\ \text{JUMP} \\ \text{STORE} \\ \vdots \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccc} A & B & C & D & \cdots \end{array} \\ \left[\begin{array}{ccccc} 1 & 0 & 1 & 1 & \cdots \\ 0 & 1 & 1 & 1 & \cdots \\ 1 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \\ 1 & 0 & 1 & 1 & \cdots \end{array}\right] \end{array}$$

Next, the <u>instruction mix</u> vector, P_I, is composed with each element equal to the probability of use of the instruction corresponding to its row. (The frequency of usage for each instruction must be known to apply this procedure.) Then, the vector P_{DP}, the data communication path probability vector is computed as $P_{DP} = P_I^T R_T$. Here each element is proportional to the frequency of usage for the corresponding communication path. From this the degradation of performance can be found if the path were removed.

Performance degradation is evaluated as follows. Let

P_i = probability of instruction i,

$N_{MR}(i)$ = number of memory references for instruction i,

$N_C(i)$ = number of strictly computation (non-memory ref.) cycles for instruction i, and

T = average time of execution per instruction,

where

$$T = \Sigma P_i [N_{MR}(i) + N_C(i)]. \tag{2.1}$$

If the removal of a data communication path, j, increases the execution time of those instructions which use it by one cycle, then, letting b_{ij} be a Boolean variable equal to one if instruction i uses path j, and zero otherwise, we have

$$T' = \sum_i P_i[N_{MR}(i) + N_C(i)] + \sum_i P_i b_{ij} \tag{2.2}$$

where T' is the new average instruction execution time. Performance degradation, $\Delta\bar{T}$, is then

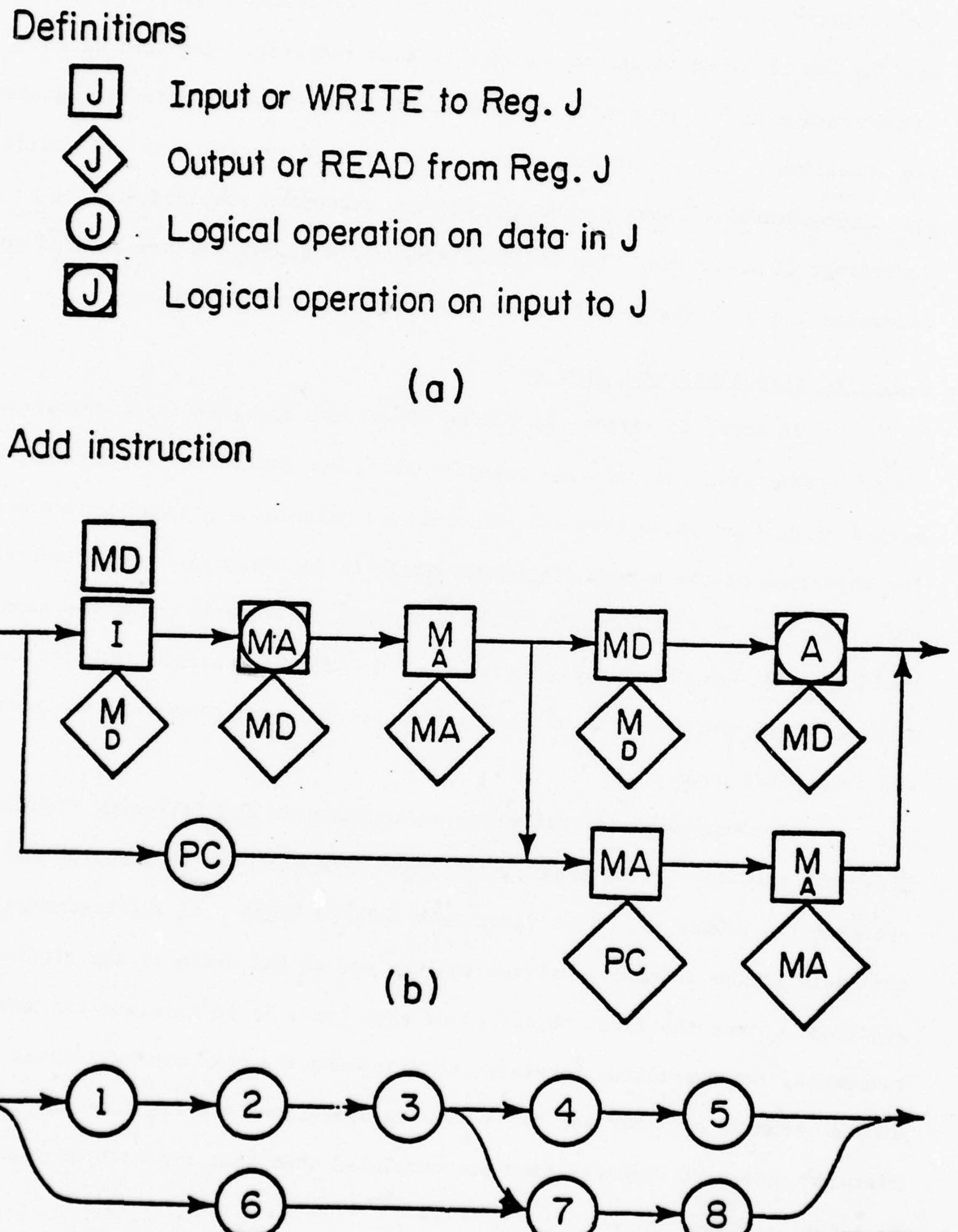
$$\Delta\bar{T} = \frac{T'-T}{T} = \frac{\sum_i P_i b_{ij}}{\sum_i P_i[N_{MR}(i) + N_C(i)]} . \tag{2.3}$$

For any path j that is removed, if the use of the instructions which use this path is small the performance degradation will be small.

2.6. Modified Precedence Graph

The register transfer graph of the previous section gives a minimal interconnection pattern but does not reveal which paths must be duplicated because they are used twice during a cycle or used in different phases of a cycle. For a single processor the register transfer graph provides all the needed information, but for a pipelined processor, when the data transfers occur at different times during a cycle, a separate data communication path and/or functional unit is required. The first step toward developing a timing graph of the processor is to develop precedence graphs for each instruction which indicate specifically the resources used at each time.

To provide this information a modified precedence graph for each instruction is formed. Definitions for the modified precedence graph are shown in Figure 2.4(a). An example of this graph is shown in Figure 2.4(b) for the ADD instruction used in Figure 2.1. The precedence graph from Figure 2.1 is shown in Figure 2.4(c). The modified precedence graph exhibits a one to one correspondence between its groups of inputs, outputs and functional operations and the functional components of the previous precedence graph.



Definitions
J Input or WRITE to Reg. J
J Output or READ from Reg. J
J Logical operation on data in J
J Logical operation on input to J
(a)
Add instruction
MD
I
MA
M A
MD
A
MD
MA
PC
MA
M A
PC
MA
(b)
1
2
3
4
5
6
7
8
FP-5691

Figure 2.4. Modified Precedence Graph.

The registers involved in each operation and data transfer are indicated as well as any required inputs or outputs to this register. Required data communication paths can easily be determined since output and input registers are specified. The use of functional units is also specified in the modified precedence graphs. These graphs specify the resources required and the precedence of their use. The modified precedence graphs can now be used to determine the time assignments of all resources in the processor.

2.7. The Time Assignment Problem

In order to relate the timing of all the register input operations, output operations, uses of data transfer paths and functional operations, a merged precedence graph from all the modified precedence graphs is composed. The objective of the merged precedence graph is to assign all operations and data transfers to time periods of the processor cycle while using the same resource (e.g. output of register MA) as few times as possible in order to minimize input-output ports of internal registers, data communication paths and functional units.

The merged precedence graph is composed in the following fashion. First the topology of the graph is defined. The graph is to describe one cycle of the processor. Time flows from left to right with the beginning of the cycle at the leftmost position and the end of the cycle at the rightmost position. Since the first objective of each cycle is to generate the memory reference, the operations necessary to determine the next memory address are done as soon as possible and all remaining operations are fit into the cycle to minimize resources provided they are completed when that resource is required by a new processor cycle.

The graph is arranged vertically with a row for each register and for the memory data and address ports. Within the graph, the same symbols defined for the modified instruction precedence graph are used to indicate register input and output operations and functional operations. However, a functional operation done along with a data transfer is indicated as a circle on the input to the result register. Double-line arcs indicate a data transfer and are drawn from the output register to the input register. Precedence arcs are indicated by a single line directed arc drawn from the input register of the preceding double arc (or single register operation) to the output register of the following double line arc (or single register operation).

To construct the graph, first the left margin is labeled indicating a row for each register and memory bus (separate for A^M and D^M). Then, the modified precedence graph for each instruction is traced. Each operation in the modified instruction precedence graph is labeled on the merged precedence graph in the same horizontal positions as operations performed at the same time during the cycle as operations from other instructions. The accuracy of the initial horizontal placement will not affect the resultant graph but it will ease reading the initial merged precedence graph. The data transfer operations (e.g. output from memory data bus and input to the MD register) require the input and output resources to be placed in the same column and connected with a double-lined arc oriented in the direction of flow of data. Any precedence relationships are then indicated with a directed arc from the preceding resource to the following resource.

A merged precedence graph has been constructed for the example instruction set and appears in Figure 2.5. When composing this graph one

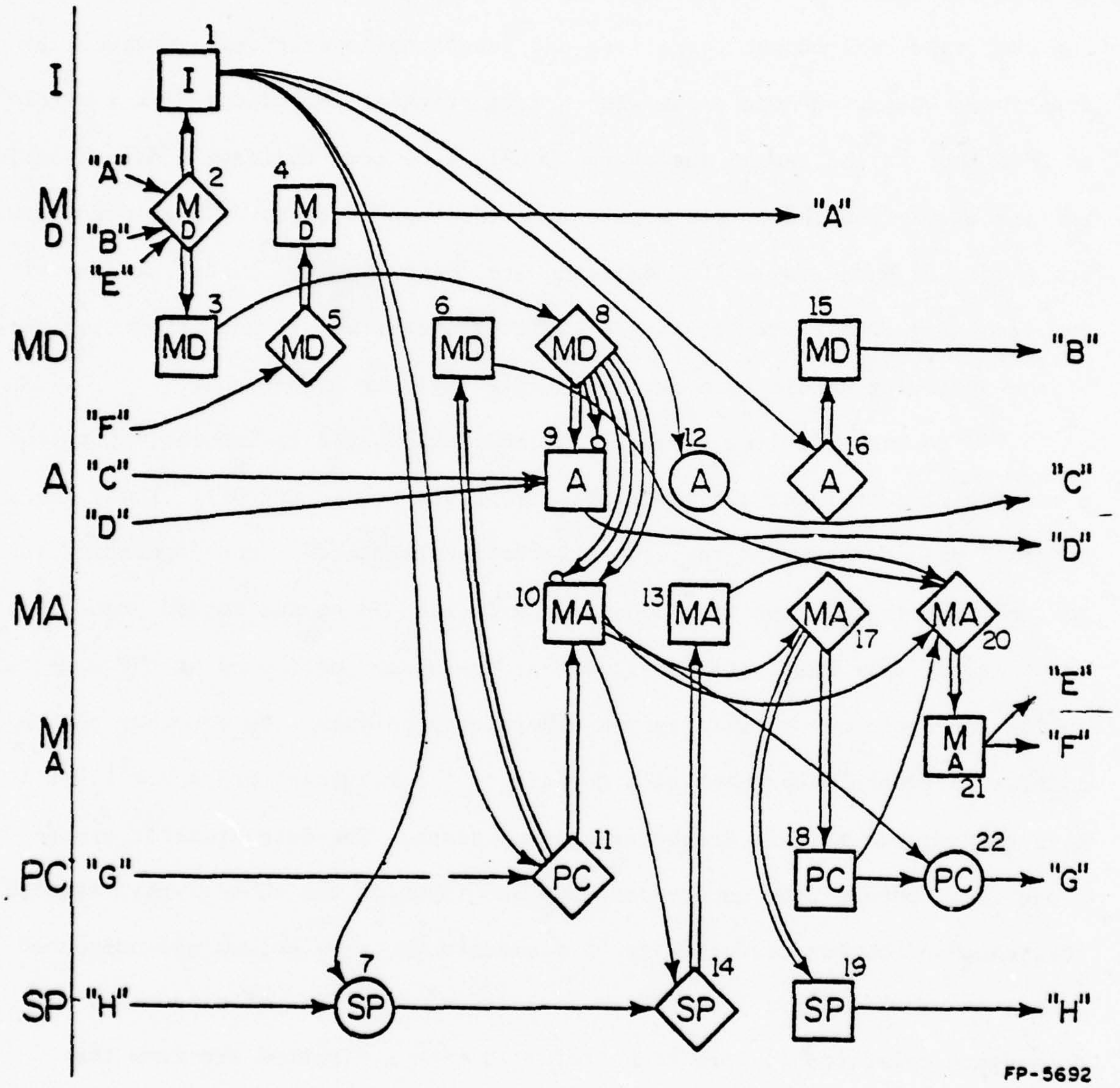


Figure 2.5. Merged Precedence Graph.

could also label the precedence and data transfer arcs with the instructions which use them, thus providing a measure of how often each path is used.

The correspondence between this graph and the final processor is that each register input and output in this graph requires an input or output port associated with that register in the final processor. Each output-input pair in the graph connected by a double line arc corresponds to a communication path in the final processor. In order to simplify the physical structure of the processor, this graph should be reduced as much as possible. To do this, merges are found whenever possible where the same input operation, output operation or data transfer used by different instruction cycles can be merged to occur during the same period of the processor cycle. This merging reduces the number of register input ports, register output ports and data communication paths required in the physical structure of the processor.

When finding merges, the most important constraint arises from the precedence relationships. It would be convenient to merge the graph so that each register has only one input and one output port (which is the minimum for each register), but the precedence arcs prohibit this when the objective is to generate a memory reference during every cycle. To determine whether two inputs or two outputs can be merged, one must examine all precedence paths involving the two input-output pairs in question. When considering an input operation (to a destination register), the precedence arc is drawn from the destination register of the prior operation to the source register that is paired with this input operation. To determine whether two operations can be merged, one must trace the precedence paths to the beginning of the cycle. If, when tracing a precedence path back from one operation, it is found that it must follow the other operation, the two operations cannot be merged. If all pairs of precedence paths (one from each operation) are traced back either to the beginning of the cycle or to a common operation, the two operations may be merged. This procedure can be followed with any pair of operations that have a possibility of being merged until all the possibilities are exhausted.

An example of the merging procedure can be shown using Figure 2.5. Examine the case of trying to merge outputs of the MA register, numbered 17 and 20 (each operation has been numbered in this figure to aid in explanation). Tracing back along the precedence arcs from operation 20, it can be seen that input 18 to the PC register needs to precede number 20. Input 18 is connected to output 17, however, by a double line arc indicating that these must be performed at the same time. Therefore output 17 must precede output 20, thereby prohibiting outputs 17 and 20 from being merged.

Another case is to examine the possibility of merging inputs 6 and 15 of the MD register. Input 6 has no precedence arcs leading to it but must occur at the same time as output 11 of the PC register. Output 11 is preceded by input 1 which occurs at the beginning of the cycle. Input 6 is thus only preceded by 1. Input 15 has no precedence arcs either but must occur with output 16 of the A register. Output 16 is preceded only by input 1. Therefore inputs 6 and 15 are both preceded by the same operation and can be merged.

It is also possible to merge an input and an output of the same register into the same time phase in accordance with the precedence relationships (assuming registers consist of master-slave or edge triggered flip-flops). Several of these merges have been performed (e.g. input 13 and output 17 of the MA register) for the example in Figure 2.5 and the final set of operations has been grouped into columns each of which represents one phase of the processor cycle. The resultant merged precedence graph is shown in Figure 2.6.

2.8. Processor Layout

The final form of the merged precedence graph can be used to give the layout of the processor. The columns of Figure 2.6 have grouped the register transfer and functional operations into a set of time slots which must occur in order in the processor. Each of these columns therefore corresponds to a phase in the pipelined implementation provided none required an excessive amount of time to be performed in which case they may be divided into two phases. The double-lined arcs define data transfer paths that are needed in the final processor. All inputs and outputs of registers are also defined by Figure 2.6. Figure 2.7 shows the layout of the processor for the example as obtained from the merged precedence graph of Figure 2.6.

A copy of each register is needed for each phase of the processor (corresponding to each column of Figure 2.6). The appropriate connection paths are added during the phase to which they were assigned in Figure 2.6. Each register is also connected from its predecessor time copy and to its following time copy. Data transfers are made by selecting the appropriate input to each register from among the previous time copies of the register and the various register and functional unit outputs connected to the selector. The selector is a simple multiplexer with control inputs from the CPU control logic. Functional operations are selected by the same method as selecting data transfers. The memory data and address busses are connected to the assigned phases as in Figure 2.6. The outputs of the register on the far right are just connected to those at the beginning of the cycle at the far

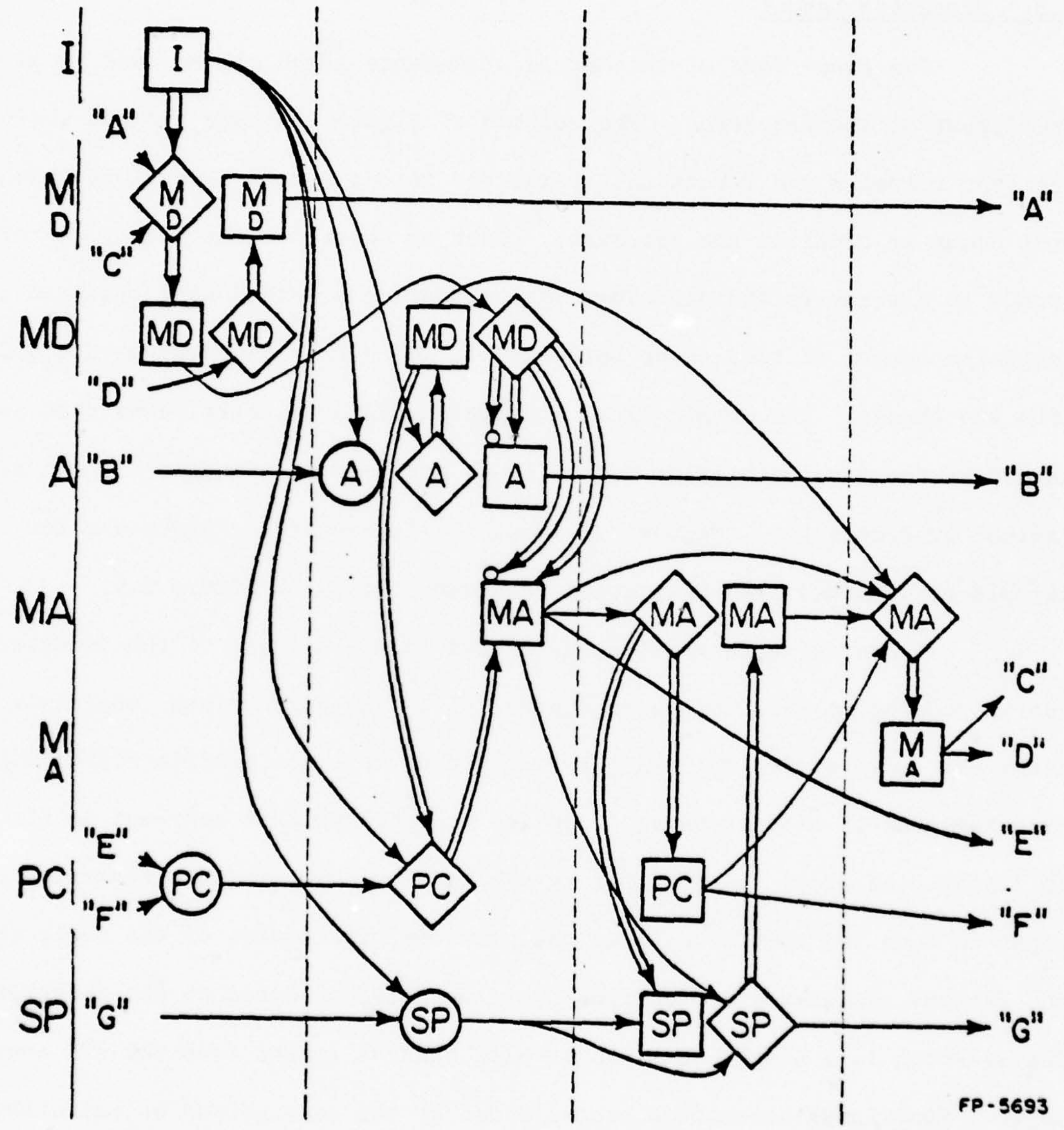


Figure 2.6. Resultant Merged Precedence Graph.

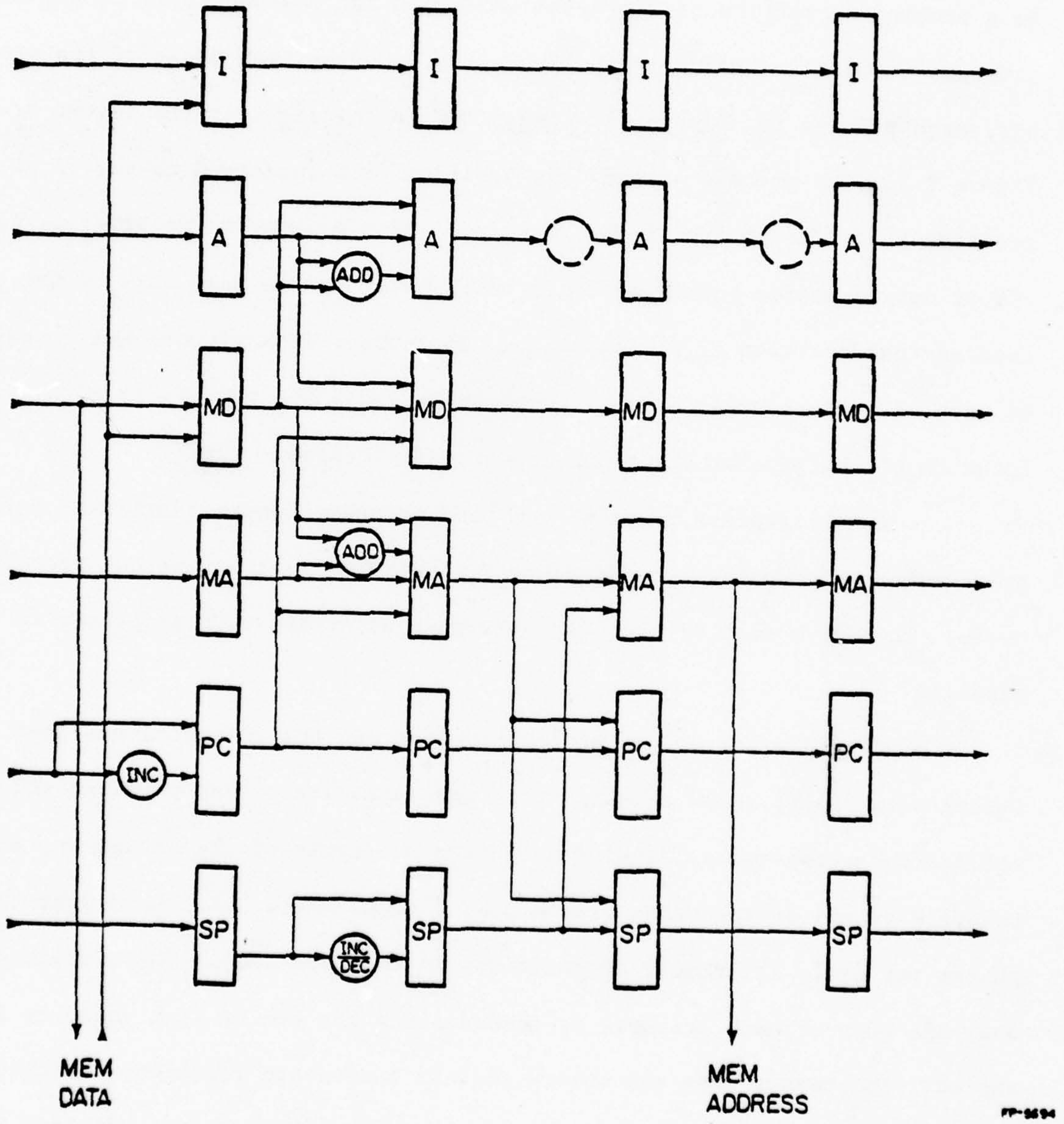


Figure 2.7. Final Layout.

left of Figure 2.7. The operation of this processor is as follows. One column of Figure 2.7 contains the information for one "process." A <u>process</u> is a program or portion of a program which can run independently of any other programs currently running in the processor. During each phase of the processor, each process is transferred one column to the right in the layout of Figure 2.7. The process at the last position is transferred to the first position at the beginning of the pipeline. Each column of the pipeline has a set of data transfer paths and logic units which can be used only by the process at that position in the pipeline. Therefore, each process must make use of each of its resources during the phase in which the process is at the location of the pipe at which the resource is located.

A multiprocess pipeline has been organized where functional units and data transfer paths are shared by several processes during one processor cycle. However a copy of each register must exist for each processor in the pipeline.

Consider one column of the pipeline and the resources assigned to it. During each phase, a new process is at that position of the pipeline and may utilize those resources. Therefore during one cycle of the processor, one resource or set of resources can be used S times where S is the number of phases per cycle (or number of positions in the pipeline). For S processes, only one copy of each resource is needed since the use of each resource is shared. The memory data and memory address busses are resources assigned to phases and can be used during each time phase by some process. If each process executes in a manner which requests one memory reference for each cycle of the processor, the memory data and address busses will be utilized every time phase, thereby obtaining maximum usage of these resources. These are the most costly and most limiting resources since they require pins and high utilization will produce high throughput for the processor.

The structure described in Figure 2.7 is a highly regular structure of registers which will reduce the area needed for interconnections within the chip. Since each of the processes running in this processor is independent and each resource can be utilized only when the process is located at its position in the pipeline, there exist no possible access conflicts (collisions) [SHA74] for the use of any resource. This elimination of conflict resolution enables simplified control of the pipeline since each process can be controlled in the same independent manner as it would be if it were running on a single microprocessor. Some specific control structures which can be applied directly to the organization are discussed in [KOG77].

2.9. Conclusions

A method of organizing a pipelined processor suitable for implementation with LSI has been described. It assumes that an instruction set is chosen which is designed to efficiently perform the intended job load of the computer. The method presented decomposes these instructions, formulates which capabilities the processor must include and creates precedence relationships among the operations of the processor. These operations are then combined when possible to reduce the complexity of processor and a final layout is obtained. Graphical constructs aid in the analysis of alterations.

The pipelined structure does not require complex control structures due to the independence requirement of the processes. The regular structure also reduces interconnection costs and complexities,which is necessary to make efficient usage of chip area in a LSI implementation.

Thus a simple structure is obtainable for a pipelined processor suitable for LSI implementation. The additional requirements needed for this processor over a single processor are the additional area for the time replications of each register and the necessary software to schedule the independent processes.

CHAPTER 3. PARTITIONING

3.1 The Partitioning Problem

When a system is too large to be implemented on one chip, it must be partitioned into small enough parts to be implemented with a set of chips. As discussed in Chapter 1, there are two constraints which limit what can be implemented on a single chip. These are the total chip area allowed and the total number of pins allowed per chip. These constraints apply to partitioning as well as single chip design.

In reality, these constraints are soft constraints but as they are exceeded, cost-effectiveness falls sharply. For use in this research, a hard limit will be assumed. When one of the two constraints are not met, a partition which reduces the chip area or pin requirements per chip must be found.

When it is determined that a partition is required, the structure of the system to be partitioned must be studied to find sets of units which have few I/O connection paths between them. It is usually easy to partition into several units with smaller area, but it is usually difficult to achieve only a small number of connections between units. With many partitions, the number of pins per chip will actually increase. Typically, the number of pins required for the partitioned chip set is significantly higher than the number of pins required if the system were implemented with one chip. Partitions are sought which minimize the number of pins required. This chapter formulates and studies several general classes of partitions and compares the effectiveness of two most useful methods of partitioning.

3.2 Approaches

There are several standard ways of partitioning a chip to reduce the required number of pins [PEN72]. The application of these standard approaches to a pipelined CPU studied as well as an approach specific to a pipelined CPU.

The first method of partitioning is called a register partition (or functional partition). The register partition divides the processor into sets of functionally related registers and functional units. Figure 3.1 shows an example of a register partition of a CPU. Here the processor is divided into three sections, the control section, the addressing section and the data section. These sections are determined by finding groups of registers which have many connection paths between them (see Figure 2.2). A partition separating these highly connected groups would require many pins. The register partition places each of these groups of highly interconnected registers on the same chip and pins are added only to provide connection paths between sections.

The second partitioning method is called a bit slice partition. A bit slice partition places several bits of each of the registers and functional units onto the same chip according to their relative bit position in the data word. Figure 3.2 is an example of a bit slice partition formed by dividing the processor into a high order half and a low order half. All components processing the low order bits of data words are placed on the low order chip and likewise for the high order chip. Bit slice partitions may be made as finely as a one bit wide slice per chip. This type of partition is good for CPU structures which have few connections between bit levels but many connections within each bit level.

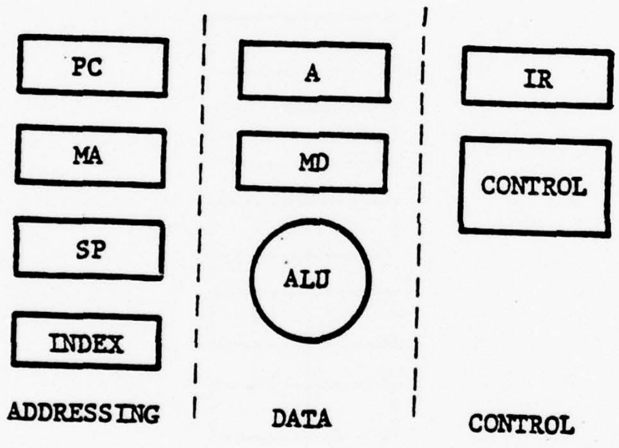


Figure 3.1. A Register Partition.

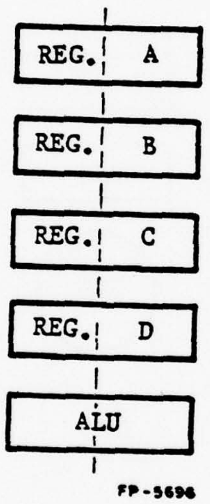


Figure 3.2. A Bit Slice Partition.

The third type of partition, called the time slice partition, places the registers and components which belong to the same phase or group of phases of the processor cycle on the same chip. An example of this method is shown in Figure 3.3. This partition is applicable only to the pipelined processor. It does not require pins for data transfer paths connecting registers and functional units assigned to the same phase of the processor cycle.

The time slice partition just described divides the processor into several sections each containing a portion of the processor cycle. This partition requires pins on each chip to communicate the signals which must be transferred from one phase of the processor to the next. Figure 3.3 demonstrates the connection paths cut for the processor described in Chapter 2. From this diagram it is seen that for each distinct register on a chip, two data word width paths are cut.

If each data word is N_D bits wide, $2N_D$ pins are required per chip per register. This is an exorbitant number of pins, e.g., for a six register CPU with a 16 bit data word, this partition would require at least 192 pins per chip! Time slicing is therefore discarded as a practical method of partitioning.

This analysis thus shows that all phase replications of a register should be placed on the same chip. This restriction reduces the many possible partitions and simplifies the analysis of the other types of partitions.

A special case of the time slice partition is a process partition where each chip contains all hardware necessary for one process. It basically results in a set of independent parallel processors. However, each

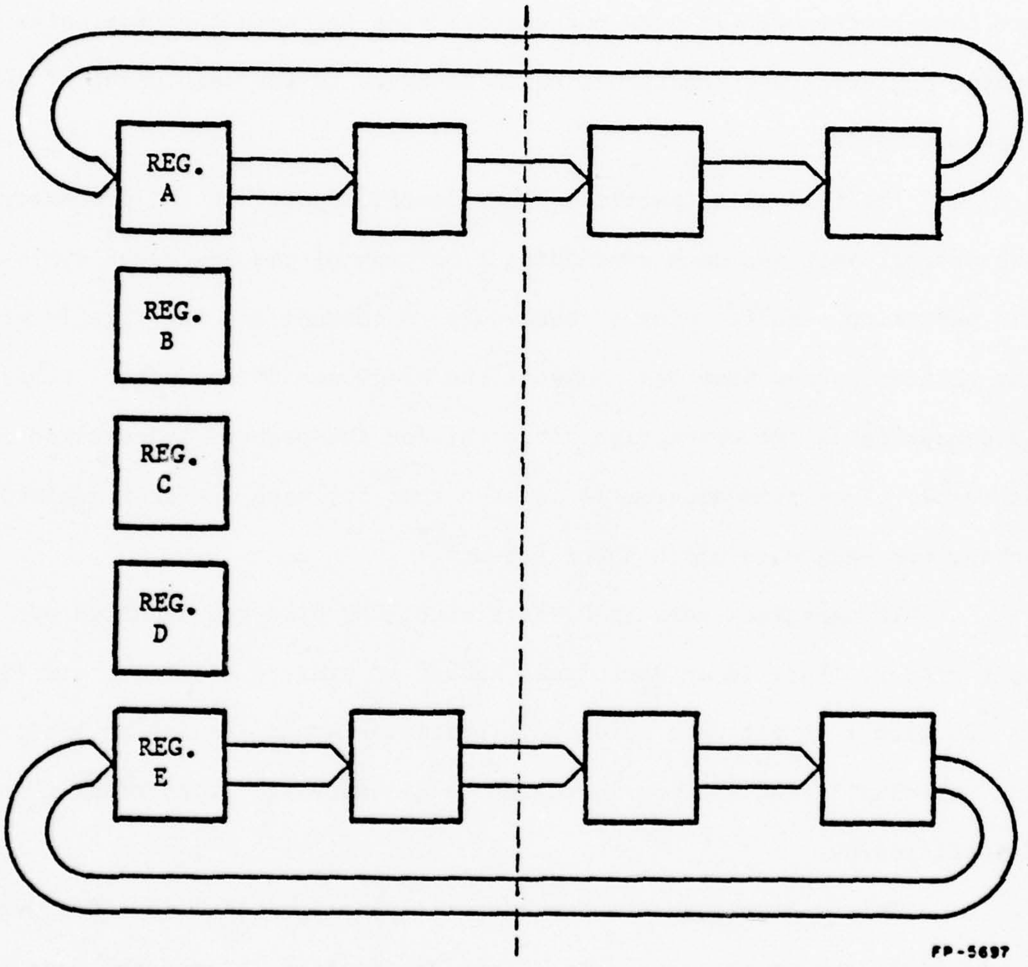


Figure 3.3. A Time Slice Partition.

process remains in its chip and does not move from chip to chip to execute a cycle. This partition is similar to a time slice for each phase with all control and functional units added to each chip, although some functional unit sharing within the chips (as in an ALU-bus organization) is possible. This partition, however, does not use pins efficiently and involves much duplicated hardware and internal interconnection area. Since this partition requires much poorly utilized hardware, it will be discarded.

Only the bit slice and register partitions are considered further.

3.3 Bit Slice Analysis

The bit slice partition is a method used for dividing the processor in many commercially produced microprocessors. The idea of the bit slice is that most data connection paths exist between registers and functional units within the same bit level of each device. Connections between bit levels are few (carries for addition and link bits for shifts are two common examples). Bit slice partitioning thus appears to be attractive. However, all control signals must be connected to each chip, since each bit level still requires all control signals. Control pins per chip are therefore not reduced. Since all control signals must be connected to each chip and the control section is not easily partitioned between bit levels, the entire control section for the bit sliced partition is generally implemented on one chip (or more) separately from the chips of the bit sliced processing section. For each bit slice partition the number of data pins in the unpartitioned chip is divided evenly among the partitioned chips, link data pins are added and all control, power and clock lines appear on each chip.

In order to analyze the bit slice partition, a model of the pin requirements is needed. Pins of the original chips must be classified and the number of link data pins determined. The classification scheme is defined below. The control section itself is not modeled.

N_D	= number of bits in the data word
N_C	= total number of control pins from control section to processor section
N_{CP}	= the number of clock and power pins required
N_{CL}	= number of carry, link and other sense pins required between bit levels
D	= number of N_D wide I/O ports to the entire CPU
P_i	= average number of pins per chip for an i chip partition
P_T	= total number of pins in entire system

Using these pin classifications, the number of pins required by the processing section if it were implemented with one chip is

$$P_1 = DN_D + N_C + N_{CP}. \tag{3.1}$$

For a bit slice partition with k chips, assuming k divides N_D and $k > 1$, the number of pins per chip becomes

$$P_k = D(\frac{N_D}{k}) + N_C + N_{CP} + N_{CL}. \tag{3.2}$$

The term in this equation which is reduced is the N_D term, i.e., the data word width per chip is reduced by a factor of k by this partition. The two terms that are not reduced by this partition are N_C and N_{CP} which represent control, clock and power. Carry and link pins are added to each chip when

$k > 1$. $N_C + N_{CP} + N_{CL}$ pins must be included for each partitioned chip and provide a lower bound on the number of pins per chip for all values of k.

The reduction in the number of pins per chip in going from a one chip implementation to a two chip bit slice partition is

$$\Delta P_{1,2} = D(\frac{N_D}{2}) - N_{CL}. \qquad (3.3)$$

The reduction in the number of pins caused by successively finer bit slice partitions is

$$\Delta P_{k,k+1} = DN_D(\frac{1}{k} - \frac{1}{k+1}), \qquad k \geq 2. \qquad (3.4)$$

This continuous approximation, assuming all k divide N_D, provides some intuition about the bit slice partitioning process. This analysis shows that in addition to the pin reduction, going from $k = 1$ to $k = 2$ incurs a penalty of N_{CL} pins, the number of pins required for carry and link information. As bit slice partitioning continues with larger values of k, Eq 3.4 shows that the pin reduction is lower since k is increasing and as k increases, the term $(\frac{1}{k} - \frac{1}{k+1})$ decreases asymptotically to zero. The following is an example of applying the bit slice partition to the data handling part of the processor used as an example in Chapter 2. The values for all variables are: $N_D = 16$, $N_C = 12$, $N_{CP} = 4$, $N_{CL} = 6$, $D = 2$. Then:

$$P_1 = 2(16) + 12 + 4 = 48 \qquad P_T = 48$$

$$P_2 = 2(\frac{16}{2}) + 12 + 4 + 6 = 38 \qquad P_T = 76$$

$$P_4 = 2(\frac{16}{4}) + 12 + 4 + 6 = 30 \qquad P_T = 120$$

$$P_8 = 2(\frac{16}{8}) + 12 + 4 + 6 = 26 \qquad P_T = 208$$

$$P_{16} = 2(\frac{16}{16}) + 12 + 4 + 6 = 24 \qquad P_T = 384.$$

Considering the same processor but with an eight bit word width instead of 16 gives:

$$P_1 = 2(8) + 12 + 4 = 32 \qquad P_T = 32$$

$$P_2 = 2(\frac{8}{2}) + 12 + 4 + 6 = 30 \qquad P_T = 60$$

$$P_4 = 2(\frac{8}{4}) + 12 + 4 + 6 = 26 \qquad P_T = 104$$

$$P_8 = 2(\frac{8}{8}) + 12 + 4 + 6 = 24 \qquad P_T = 192.$$

In the first example a reasonable reduction was obtained for the first two partitions, but by the third partition there was a reduction of only four pins per chip. The total number of pins for the entire system is also interesting. They increase drastically as partitioning continues. The whole system cost increases with the total number of pins indicating that whenever possible partitioning should be avoided and used only to the extent necessary when required.

In the second example, when the data word width is small causing very little reduction in pins per chip to be achieved by any of the partitions. Again, the total number of pins increases rapidly as in the first example. In fact, by the third partition there is only a difference of 16 total pins out of 200 between the wide word case and the small word case. This indicates that many duplications of pins are needed at this high degree of partitioning.

In summary, the bit slice partition reduces data pins per chip with each partition but there is an initial penalty for partitioning and the amount of pins reduced decreases as partitioning increases. The bit slice partition gives good results for the first several partitions of a processor with a wide data word but does not do very well when the data word width is small with respect to the number of control pins. The total number of pins for the system increases rapidly with increasing partitioning. Bit slice partitioning, therefore, should only be performed if absolutely necessary since the system implementation pin cost is great with a high degree of partitioning.

3.4 Connection Graph Model

The register slice or functional partition divides the processor by placing related sets of entire registers and functional units on separate chips. The pins required for each chip in the register partition are the power and clock pin, the control pins needed to carry only those control signals for the units on each particular chip and the pins that send the data to and from the set of registers and functional units on each chip. With this partitioning method, it is not so straightforward to determine the best partition as with bit slice partitioning. A partition is needed that separates the processor into sets with a minimal number of communication paths between them.

A graphical representation of the processor aids in understanding the connection pattern between registers and functional units and in finding good register partitions. This graphical representation, called a connection graph, is described here and will be used later to find register partitions.

The connection paths in a processor layout are composed of control signal lines and data path lines. The data path lines however consist of N_D (= number of bits in the data word) lines in parallel. Most processors have data word widths of at least 8 or 16 bits. The number of connections required for the data path lines to each register and functional unit are usually much greater than the number of connections for control lines to each register and functional unit. The choice of a good register partition is thus influenced mainly by data lines. Therefore, when constructing the connection graph, the control lines are omitted. If a more detailed connection pattern is required, the control lines may be included, but their omission eases analysis without significantly affecting results.

In order to graph the pipelined processor of Chapter 2, a simplification can be made due to the results of Section 3.2. There it was found that any partition that separated time replications of registers would require an excessive number of pins and would thus be impractical. Therefore all phase replications of a register will be placed on the same chip and for partitioning purposes each register need only be modeled by a single register copy. Among the register connection paths, only those that connect different registers need be modeled for partitioning.

The processor can thus be modeled using one vertex for each register in the processor and an edge of weight one between each set of registers for each data communication path between them. Using this model the graph of the processor of Chapter 2 is shown in Figure 3.4.

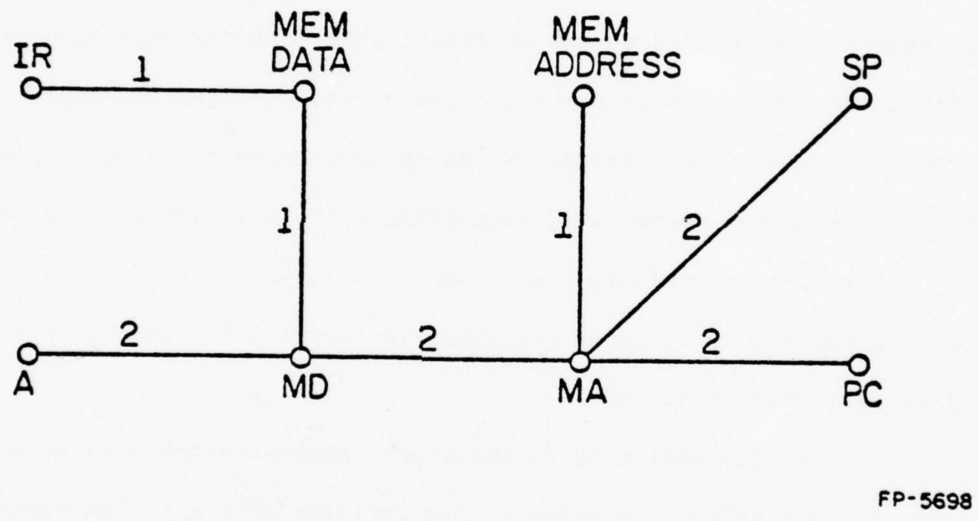


Figure 3.4. Connection Graph of Processor.

The functional units are not represented in this graph since their outputs are connected only to one register and have been grouped with that output register. This graph is basically a simplification of Figure 2.2 but includes any connection paths added after the time assignment of each resource has been made. This graph includes only the connection paths and registers, which is all the information needed for register partitioning.

After some study of this graph it becomes apparent that it has some deficiencies. The situation where there is a "Y" in the communication path is not accurately modeled. A "Y" in the connection path occurs when an output is connected to two or more inputs. If the case occurred that both input vertices were to be grouped on one chip and the output vertex on another, the obvious procedure would be to use one connection from the output vertex to the second chip containing both input vertices and dividing the connection path within the second chip. This connection method cannot be obtained easily by using the graph in Figure 3.4. Therefore a somewhat different model is needed.

An alternative is to model each communication path as a vertex also, and use an edge to connect each register vertex to the communication path vertex and another edge to connect the communication path vertex to the next register vertex. Thus a communication vertex can be assigned to a chip along with one of the two register vertices showing only one I/O data connection path into or out of the chip. This modification produces the graph shown in Figure 3.5.

This modification gives a model which will lead to fewer communication pins per chip but in its present form, Figure 3.5 is unnecessarily

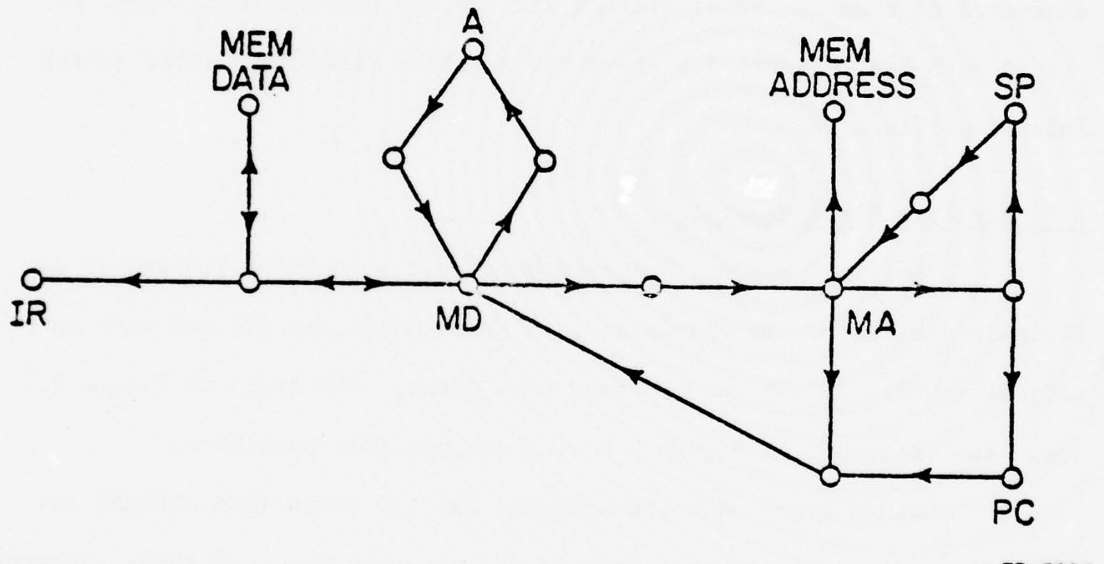


Figure 3.5. Extended Connection Graph.

complex. Where a vertex is used to represent a communication path which involves data transfer between only two registers, there are only two edges connected to it. Adding the communication vertex to one chip or the other (provided the two registers are on different chips) does not change any connection requirements. Therefore the communication path vertex can be removed, thence requiring only one edge between the two corresponding register vertices. Thus all communications vertices with only two edges connected to them can be eliminated and the two corresponding edges replaced with one. Figure 3.6 shows the graph of Figure 3.5 after making this simplification.

3.5 Register Slice Analysis

The final connection graph developed in Section 3.4 has an arc for each data path. By examining this graph model one can see ways of partitioning which do not cut too many data paths. The graph in Figure 3.7 shows the processor of Figure 3.6 with one possible partition.

Dashed lines show the original two I/O buses (MEM ADDRESS and MEM DATA) and a register partition which cuts only one data path. However, this partition may not divide the area properly and a different partition may have to be found.

The connection graph can be used to examine various other possibilities and to determine how many data paths will be cut in each case. Good partitions result when few data paths are cut and the implementation area needed by each component of the partition is reasonable. Furthermore, good cut sets often divide the system between existing input output buses. In the processor example above, since the system has only two existing I/O

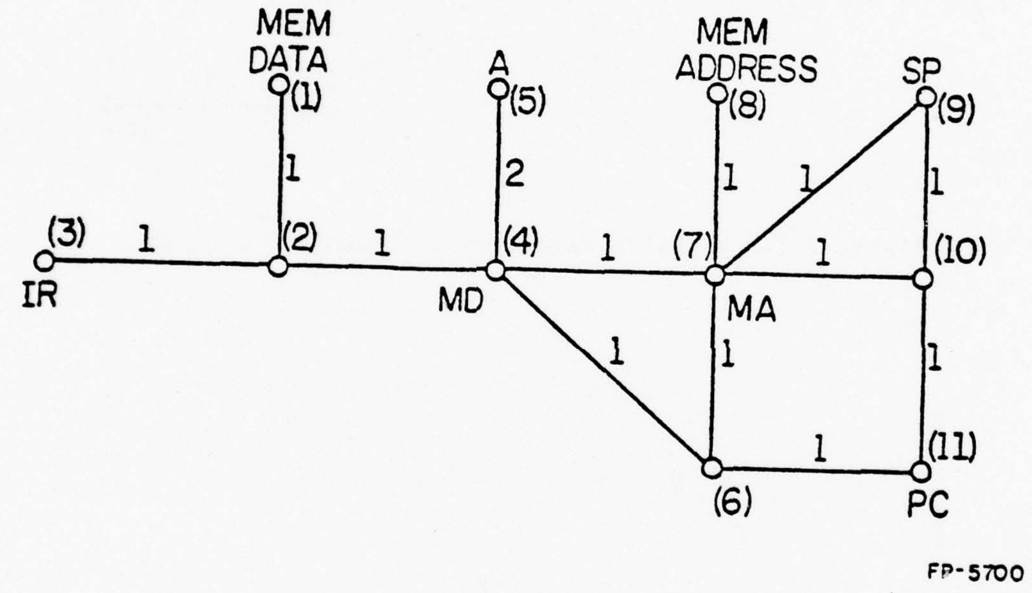


Figure 3.6. Simplified Extended Connection Graph.

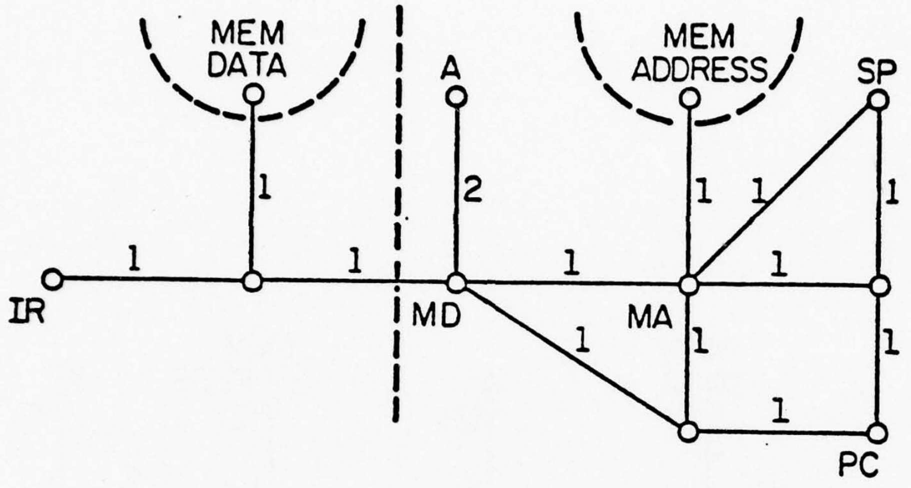
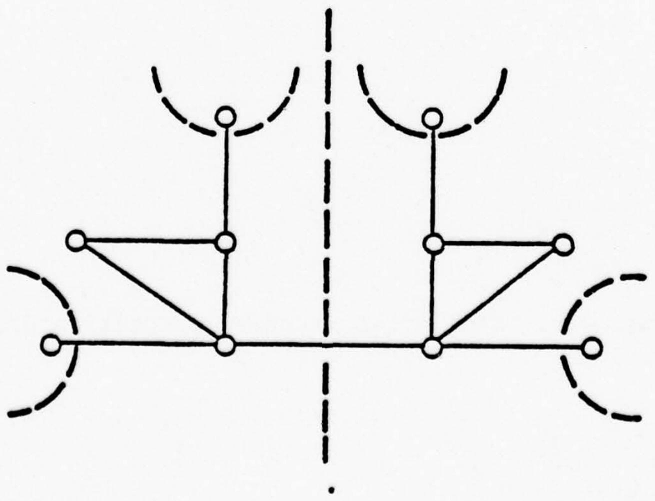


Figure 3.7. Register Partition of Example Processor.



FP-5701

Figure 3.8. Partition of a Special Four Bus Chip.

buses, any partition that does not divide the chip implementation between the two buses involves one chip with at least three I/O connection paths, two for the existing buses and at least one more bus to connect the chip with some other chip (provided the original connection graph was a connected graph). Other chips could actually have more data pins than the original single chip implementation. When a substantial increase in the number of data pins occurs, it is difficult or impossible to make up the difference by reducing other types of pins.

A distinct advantage of the register partition lies in the fact that partitioning between existing I/O buses can reduce the number of I/O connection paths per chip significantly. A special example of this is shown in Figure 3.8 where a four bus chip is partitioned into two three I/O connection path chips. However this type of reduction is only possible in special cases.

In order to analyze the effects of a register partition numerically, a model similar to the one used for the bit slice partition in Section 3.4 is used. The following definitions apply:

N_D	= number of bits in data word
N_C	= number of control signal pins
N_{CP}	= number of clock and power pins,
C_{P_n}	= number of data communication paths cut by a partition into n chips
n	= number of chips in partition
D	= number of I/O buses
P_n	= average number of pins per chip for an n chip partition

P_T = total number of pins in system

$\Delta P_{n,m}$ = reduction of pins per chip from n to m chip partitions

To be consistent with the bit slice partition the control section will be assumed to be separate from the processing section. If the control section were included with one chip of the processing section, the control pins necessary to connect control signals to the rest of the processing section would be added to the chip with the control section.

For each path cut in a partition, two additional I/O communication paths are created which require N_D pins each in the total system. This case applies except for dividing paths connected to a "Y", in which case one must consider the number of chips the "Y" bus is divided among. For v chips, v I/O connection paths are required. For each normal path cut, $2N_D$ pins are added to the system. For cutting of a "Y" path, vN_D pins are added to the system. Thus the "effective" number of paths cut is one in the former case and v/2 in the latter case. In the following, C_{P_n} represents the total effective number of paths cut by an n chip partition. The number of pins added to the system (among all chips) by cutting of paths is $2C_{P_n}$.

The equation for the number of pins required in a one chip implementation, as before, is

$$P_1 = DN_D + N_C + N_{CP}. \tag{3.5}$$

If a register partition is performed by dividing into n chips, the average number of pins per chip is

$$P_n = \left(\frac{D}{n} + \frac{2C_{P_n}}{n}\right) N_D + \frac{N_C}{n} + N_{CP}. \tag{3.6}$$

This formula only applies to the average number of pins per chip, since the original number of I/O buses and number of control pins can only be divided approximately evenly amongst the chips. Furthermore, it is assumed that the partition does not interfere with the compact encoding of control signals, i.e., that the N_C pins can be divided among the chips without any extra pins being required. By examining Equation 3.6, one can see that the terms DN_D and N_C are reduced by a factor of n, but there is an average increase of $(2C_{P_n} N_D)/n$ pins per chip. The term $(2C_{P_n} N_D)/n$ represents the fact that each effective data path cut by a partition requires $2N_D$ pins. This increment is averaged over the n chips. The register partition is thus most effective when the number of control lines is large compared to the data word size N_D. However, effectiveness also requires low values of C_{P_n}.

Under these assumptions the net reduction of pins per chip when going from n chips to n+1 chips ($\Delta P_{n,n+1}$) is found as follows:

$$\Delta P_{n,n+1} = P_n - P_{n+1} \tag{3.7}$$

Using Equation 3.6 for n and n+1 and substituting into Equation 3.7 gives

$$\Delta P_{n,n+1} = \left(\frac{D+2C_{P_n}}{n}\right)N_D + \frac{N_C}{n} - \left(\frac{D+2C_{P_{n+1}}}{n+1}\right)N_D - \frac{N_C}{n+1}. \tag{3.8}$$

Combining terms gives

$$\Delta P_{n,n+1} = \left(\frac{D+2(n+1)C_{P_n} - 2nC_{P_{n+1}}}{n(n+1)}\right)N_D + \frac{N_C}{n(n+1)}. \tag{3.9}$$

Let $\Delta C_{P_{n,n+1}}$ be defined as $C_{P_{n+1}} - C_{P_n}$. Then

$$\Delta P_{n,n+1} = \left(\frac{D+2C_{P_n}}{n(n+1)} - \frac{2\Delta C_{P_{n+1}}}{n+1}\right)N_D + \frac{N_C}{n(n+1)} . \qquad (3.10)$$

It can be seen from Equation 3.10 that if $\Delta C_{P_{n+1}}$ is too large, the actual number of pins will increase instead of decrease, indicating a poor partition. In order for the number of pins to decrease, $\Delta P_{n,n+1} > 0$. The range of values of $\Delta C_{P_{n+1}}$ which satisfy this is

$$\Delta C_{P_{n+1}} < \frac{1}{2n}\left(\frac{N_C}{N_D} + D + 2C_{P_n}\right). \qquad (3.11)$$

Equation 3.11 must be satisfied in order for the partition to be practical. If there is no partition for which Equation 3.11 is satisfied, a different partitioning method should be used.

The trends involved may be demonstrated by using the example in Section 3.4. The values for the parameters are: $N_D = 16$, $D = 2$, $N_C = 12$, $N_{CP} = 4$. The number of pins for a one chip implementation is

$P_1 = 2 \times 16 + 12 + 4 = 48$ $\qquad P_T = 48$

If $\Delta C_{P_2} = 1$ for $n = 2$ then $C_{P_2} = 1$ and

$P_2 = (\frac{2+2x1}{2})\ 16 + \frac{12}{2} + 4 = 42$ $\qquad P_T = 84$

If $\Delta C_{P_3} = 2$ for $n = 3$ then $C_{P_3} = 3$ and

$P_3 = (\frac{2+2x3}{3})\ 16 + \frac{12}{3} + 4 = 50\ 2/3$ $\qquad P_T = 152$

If $\Delta C_{P_4} = 2$ for $n = 4$ then $C_{P_4} = 5$ and

$P_4 = (\frac{2+2x5}{4})\ 16 + \frac{12}{4} + 4 = 55$ $\qquad P_T = 220$

For this example, pins decrease on the first cut when $\Delta C_{P_2} = 1$, but for further cuts, $\Delta C_{P_n} = 2$ and the number of pins per chip increases with each partition. Thus, for the example used here, the register partition is not feasible for reducing pins per chip after the first cut.

There is a way of reducing the pins of a four chip partition below that of either a bit slice or register slice partition. This is accomplished by doing both one register partition and one bit slice partition. Each partitioning method is thus used at a stage when they are most effective. The results of the register slice, bit slice and combination of both, applied to the example processor used in this research for two cases with $N_D = 16$ and $N_D = 8$ are listed in Table 3.1.

In the previous example, the particular system used had $\Delta C_{P_n} = 2$ after $n = 2$. The ΔC_{P_n} term then dominates for such a simple system as n grows. It is useful to examine the trend for systems in which cuts are possible with ΔC_{P_n} small. Taking the values of N_C, D and N_{CP} as in the previous example but assuming $\Delta C_{P_n} = 1$ for the first three cuts gives:

$$P_1 = 2 \times 16 + 12 + 4 = 48 \qquad P_T = 48$$

$$P_2 = \left(\frac{2+2x1}{2}\right) 16 + \frac{12}{2} + 4 = 42 \qquad P_T = 84$$

$$P_4 = \left(\frac{2+2x3}{4}\right) 16 + \frac{12}{4} + 4 = 39 \qquad P_T = 156$$

This gives a continued successive reduction of pins per chip as n grows in the meaningful range. However, as in the bit slice case, the reduction per cut decreases and the total number of pins required increases rapidly. These results, for both $N_D = 16$ and $N_D = 8$ are listed in Table 3.2 with the results of the bit slice and combination partitioning methods.

Table 3.1. Partitioning Results for Example Processor

Number of Chips	$N_D = 16, D = 2$			N = 8, D = 2		
	Bit Slice	Reg Slice	Combination	Bit Slice	Reg Slice	Combination
1	48	48		32	32	
2	38	42		30	26	
3		50 2/3			29 1/3	
4	30	55	29	26	31	21
5		57 3/5			32	
6		59 1/3			32 2/3	

Table 3.2. Summary of Partitioning Results for Various Machines

No. of Chips	$N_D = 16, D = 2$			$N_D = 16, D = 4$			$N_D = 8, D = 2$			$N_D = 8, D = 4$		
	Bit Slice	Reg Slice	Comb	Bit Slice	Reg Slice	Comb	Bit Slice	Reg Slice	Comb	Bit Slice	Reg Slice	Comb
1	48	48		80	80		32	32		48	48	
2	38	42		54	58		30	26		38	34	
4	30	39	29	38	47	37	26	23	21	30	27	25
8	26		20	30		28	24		18	26		22

In this table it can be seen that for the first two systems with a wide data word ($N_D = 16$, $D = 2$ and $N_D = 16$, $D = 4$), the bit slice provides a better partition than the register partition. However, the combination does better when the degree of partitioning is high, namely for the four and eight chip implementations. For the second two systems with a small data word ($N_D = 8$, $D = 2$ and $N_D = 8$, $D = 4$), the register slice performs better than the bit slice partition. This is assuming $\Delta C_{P_n} = 1$ for each partition. If $\Delta C_{P_n} > 1$, the register partition would not be practical in this example. Again, the combination of the two types of partitions performs best with a four or eight chip implementation.

3.6 Summary

The detailed analysis of the bit slice and register partitioning methods shows the following results. In both the register and the bit slice partitioning methods, the effectiveness of the partition is reduced as the degree of partitioning increases. This fact leads to using a combination of the two methods when a high degree of partitioning is required, to take advantage of the maximum reduction of pins by each method. Indeed, in the four examples shown, the combination did outperform the others. once the number of chips was four or more.

Another fact about both partitioning methods is the rapid increase in the total number of pins required to implement the system as the number of chips increases. This indicates that partitioning is very costly and should be used only when necessary and only then to the required degree. The cost of implementing a larger area chip to contain the unpartitioned system must be very high in order for the system cost to be lower with a partitioned system.

The strongest point in favor of a bit slice partition is its performance with wide data words. This occurs since the bit slice divides the processor between bits of the data word, reducing the data pins per chip, but requiring control pins to be duplicated on all chips and requiring extra pins for link and carry information.

The register partition divides the processor into functional sets. This reduces the number of control pins needed per chip but increases the number of pins for data paths between chips. Systems with small data words, but a large number of control lines, perform better with a register partition provided register partitions with small C_{P_n} are found. Register partitions can be difficult to find, however, and sometimes a practical one is impossible to obtain. Analysis of the connection graph to find partitions with very few cut data paths is required. When it is necessary to cut more than one or two data paths, register partitioning becomes unattractive.

In conclusion, partitioning should be avoided if possible, but if necessary, a bit slice should be used if the data word is large and a register partition should be used if the number of control lines is large compared to the size of the data word. When a high degree of partitioning is required, a combination of both methods should be used. If partitioning is necessary, but no attractive partitions exist, the machine instruction set and its implementation may be modified to allow for better partitions. Improving the register partitions by this means is apt to prove more satisfactory than trying to improve bit slice partitions.

Thus, for simple processors, it has been shown that the multiple instruction stream pipeline architecture is attractive in its use of area and pins. Partitioning when necessary to reduce pin count or area per chip may be successful to a limited degree using the bit slice, register slice or combination methods. Using several separate simple single stream processors is wasteful of both area and pins. The next chapter considers one comparison of a simple multiple stream pipelined processor with a more complex single stream processor.

CHAPTER 4. EFFECTS OF REGISTER SET SIZE ON PERFORMANCE AND COST

4.1 Performance Tradeoffs between a Single Stream Processor and a Pipelined Processor

The pipelined multiple process CPU is effective in obtaining high utilization of the memory port pins, but to improve performance it must not greatly increase the number of memory references needed per program over that of a single stream multiple data register processor. The additional data registers in the single process CPU reduce the number of memory references required, which decreases the execution time of a process. The decreases in references and time are proportional only if execution time is proportional to the number of memory references required. Otherwise the system is not memory limited and execution time is less sensitive to the number of memory references. For our analysis we assume the worst case for multiple process CPU's: that the systems are memory limited.

The pipelined processor executes several programs concurrently, with each generating one memory reference each processor cycle. It essentially replaces idle time of the memory port between memory references with references from the additional processes. Since the pipelined processor works with multiple processes, a copy of the register set must be contained in the processor for each process in execution. The pipelined processor requires chip area to implement these registers. It is thus important to use a minimal register set to reduce the area needed for these registers to a minimum.

Since each instruction is structured to issue one memory reference each processor cycle, the total execution time, under the memory limited assumption, is approximately equal to the number of memory references multiplied by the processor cycle time. The pipelined processor has a separate process issuing a memory reference each phase of the processor cycle. This increases the number of memory references per unit of time, and hopefully the system performance, over that of a single stream processor. There is, however, a degradation of performance due to the reduced number of data registers in the pipelined processor. This degradation of performance is evaluated by the analysis in this chapter. However, the memory limited assumption favors multiregister processors since it assumes that even they can execute fast enough to issue a memory reference each cycle.

The high use of memory by the pipelined processor will increase the memory conflict problem. However, it has been shown [BRI77] using a pipelined interleaved memory structure that the performance degradation due to memory conflicts can be made negligible.

The remainder of this chapter describes an analysis of the register usage pattern of some typical programs to determine the number of memory references issued for various register set sizes. The results are used to determine the performance of a pipelined multiprocessor and a single stream multiregister processor. Performance vs cost comparisons are made between these two processors using the memory reference results.

4.2 Performance Determination

Processor performance is now developed as a function of the number of data registers, R, available to a single stream and the number of instruction streams, S, concurrently in execution. The system performance, $P_{R,S}$, is expressed as

$$P_{R,S} = P(R)S \tag{4.1}$$

for a system with R data registers per stream and S segments of pipelining. P(R) is the performance of a single processor with R registers. To get $P_{R,S}$, P(R) is multiplied by the number of pipeline segments S. Pipelining increases the performance by S since instead of one program executing in the typical fashion in the processor, the processor cycle is divided into S segments each containing a separate process. Each process flows through the pipe in one normal processor cycle time. Therefore system performance is single stream performance multiplied by S. Each process is independent of the results of and the hardware used by any other process in the pipe. No dependencies or hardware resource conflicts occur within the processor and P(R) is not affected by S. This model assumes that memory is designed for minimal access conflict.

In order to evaluate P(R), the execution time of a processor with R registers, denoted T(R), is determined, where

$$T(R) = \frac{1}{P(R)}. \tag{4.2}$$

The overall performance of the system can be evaluated with the effective execution time of the program denoted $T_e(R)$ where

$$T_e(R) = \frac{1}{P_{R,S}} = \frac{1}{P(R)S} = \frac{T(R)}{S} . \tag{4.3}$$

The effective execution time for a pipelined processor is equivalent to the time it would take to execute a program on the pipelined processor if it could be divided into S independent processes of equal execution time and run concurrently on the pipelined processor. Alternatively, it may be viewed as the time to execute S processes on an S-way pipelined processor divided by S.

Since, as described in 4.1, the execution time is proportional to the number of memory references generated, an analysis of the number of memory references required as a function of the register set size is needed. It is difficult to determine the number of memory references saved by increasing the number of registers beyond the number allowed to be referenced in the instruction set. This would require keeping track of all the data in memory. However, it is possible to evaluate the performance degradation of a process, coded to refer to many data registers, as the register set size is reduced. Some work concerning register usage has been done [LUN77],[LUN74], but this work does not give the necessary information to evaluate the degradation of performance as a function of the number of registers available. It does, however, indicate that processor performance would not be degraded significantly by a reduction in register set size. A similar study for a stack machine is described in [BLA77]. Therefore, an analysis of register set sizes smaller than that incorporated in a large machine would be valid for finding an optimal performance/cost tradeoff.

4.3 Program Trace Analysis

In order to determine the degradation of performance as the data register set size is reduced, several program traces from the IBM System 360 were analyzed. The System 360 was used since it has a larger register set size than any microprocessor available as yet and program trace data was readily available. The register reference pattern was studied to determine which register contents would be stored temporarily in memory as the register set size is decreased. This analysis was performed assuming an optimal dynamic mapping of referenced registers to memory and physical processor registers.

For a specific register set size, each time data has to be stored in memory due to the lack of enough processor registers, a register fault occurs. The total number of register faults for each register set size is totaled for the entire program trace analyzed. The total number of System 360 memory references generated in the program trace is also totaled. Then for each register set size the number of register faults per memory reference is computed. The run time of a process is taken to be the number of cycles required to execute the process and the number of cycles is estimated as the number of memory references required, following our "memory limited" performance assumption. The number of instructions was not used as the normalizing factor since the number of register faults would vary among instructions, different programs have different instruction utilizations and time of execution of various instructions also varies.

Once the number of register faults is known, the degradation of performance is determined by assuming each register fault requires two additional memory references, one to store the data from some processor register to memory and one to load that register from memory with the referenced register's data. Just two cycles are needed since it is assumed that registers are mapped into dedicated memory locations addressable by register tags of 360 programs and no address computation cycles are needed. The register to be overwritten is selected using a variation of Belady's [BEL66] optimal dynamic paging strategy which overwirtes (reassigns) that processor register which will be next referenced furthest into the future.

The execution time for a processor with R registers is computed as two times the number of register faults, F_R, plus the total number of memory references, N_{MR}, all divided by the total number of memory references.

$$T(R) = \frac{2F_R + N_{MR}}{N_{MR}} \tag{4.4}$$

This formula is normalized to the number of memory references required by a processor with all 20 System 360 registers. This provides a measure of the performance degradation relative to a standard processor with a large number of registers.

This optimal dynamic allocation of data registers will give the fewest number of faults for each register set size. This scheme actually favors the results for large register sets since optimal allocation is more easily reached in practice for small register set sizes. This is especially true for a small processor where the code generation software is not as sophisticated as a 360. The instruction set of the 360 is also more powerful and able to use a larger set of registers more optimally. The more sophisticated instruction set also requires fewer fetches which makes data memory references a more significant portion of the total number of memory references. All these points will cause the results of this analysis to be biased toward favoring larger data register sets than would actually be useful in a small processor.

One problem with the analysis performed is that some of the total of 20 registers are used for addressing purposes. Other analyses of the same traces used here was performed by Hammerstrom [HAM77]. This anaylsis has shown that on the average only three or four registers are used for addressing leaving about 16 for data usage. This effect does not cause inaccuracies in the evaluation data for register set sizes less than 15 or 16, which is sufficient for our purposes. Also, four registers of the 360 are dedicated for floating point operations and similar data in an LSI processor as discussed here would be kept in the same register set and be interchangeable with other data, as contrasted with the 360. Our analysis has assumed that the floating point registers are interchangeable with fixed point registers as in the LSI processors. Any inaccuracies due to this effect would be caused by larger register sizes being required in some cases in the 360 and multiple registers in some cases would be desirable if an LSI processor if high precision arithmetic is needed in the LSI processor.

The analysis data obtained here should thus be reasonably accurate if it is assumed that an LSI processor has at least enough registers of sufficient word width to execute any single one of its basic instructions without register faults, if its registers are allocated properly at the beginning of the instruction. Given this assumption, other assumptions which affect the accuracy of our results cause a consistent bias in favor of large R processors over small R processors.

4.4 Program Analysis Procedure

Program traces were generated for three programs, two doing numerical analysis computations and one, list processing. The first program, GAUS, which performs a Gaussian elimination on a 20 x 20 matrix to solve a set of simultaneous linear equations. GAUS was written in Fortran IV. The second program, EIGEN, is a Fortran numerical analysis program which determines the eigenvalues of a 14 x 14 matrix. This program was taken from the Eigensystem Subroutine Package of the National Activity to Test Software project. The third program, LIST, is a list processing program which first builds list structures then examines the tree structure of the data and determines the set of all accessible nodes for each sublist head.

After a program is traced, the first step in the analysis is to remove the instructions that deal with addressing since only data register usage is to be studied. A filter program was used with the filter algorithm described in [HAM77]. This provides a trace including only data register usage instructions. Next, the sequence of references to data registers made by the program is determined from the filtered trace. This sequence of register references is used to determine optimal register assignment for each register set size from 1 to 20 using a stack algorithm as in [BEL66]. From the register assignment, the number of register faults required is totaled for each register set size. The total number of memory references in the original trace is totaled and used to normalize the number of memory references required by the program for each register set size. The effective execution time for each value of R is computed by formula 4.4. The performance is inversely proportional to T_e.

Programs were written to perform the above tasks. The algorithm used to determine register faults follows.

1. Obtain the data register reference sequence from the filtered program trace, recording which register is referenced and whether a read or write reference is made to that register.
2. Read each register reference from the sequence from step 1. Number them in order starting at the beginning of the sequence. Then reverse the sequence placing it in a file.
3. For this step a table is constructed which records, while scanning the reversed sequence, the number of the last reference for each register. Each entry in the table is initially set to infinity. The reversed sequence is read. As each register reference is read, the number of this register reference is placed in the table position corresponding to this register and the reference in the trace is numbered with the overwritten number from the table. The modified reverse trace is stored.
4. Reverse the sequence from step 3 to form a forward trace. This gives the original register reference sequence but associated with each register reference is the number of the next reference to this register.

A pseudo-stack is used for steps 5 and 6. The basic operations used are the conventional stack "pop" and an "insert" operation: for which an item may be inserted anywhere in the stack and items below it, pushed down. The stack can have up to 20 entries (one for each register in the 360). The top of the stack represents that register, among the set of registers already referenced at least once so far, which will be referenced again soonest in the sequence. At each point in time the "stack" is maintained so that the registers are in sequence in increasing order of the number of their next reference. (The number of each register's next reference was stored with each reference in step 3.) Along with each register in the stack, the greatest depth attained by that particular register since its last reference is recorded with the stack entry.

5. Read in each reference in sequence from the trace generated at step 4. If the referenced register is in the stack at all, it must be at the top of the stack. In this case, add one register fault to each register set size that is less than the "depth" that this register had reached in the stack since its last reference. Do this only if this reference was a read from the register. (If it were a write, the old data needn't have been kept and a register fault would not occur.) Insert this register in the stack in the appropriate position and set the "depth" of this register to its current position in the stack.

   If the referenced register is not in the stack, then insert it into the stack according to the number of its next reference, set its "depth" to this position in the stack and increment the "depth" of any register below this register which now is in a deeper position in the stack than its current value of "depth."

6. Repeat step 5 for all references in the sequence from step 4.

This gives the number of register faults for each register set size from 1 to 20.

The results obtained from these operations give a fairly accurate measure of the register usage in the 360. However, there are some instructions that use registers for both data and addressing procedures, and it is difficult to distinguish between these during the filtering operations. The error involved should be small and only if the conclusions reached are sensitive to small variations in the numerical results (which, as it turns out, they are not) would the error be significant.

4.5 Program Results

Table 4.1 lists the number of references to each register for the three programs analyzed. As could be expected, certain registers were used much more often than others. This is especially true for the numerical analysis programs, where registers 17 through 20, the floating point registers, are used very extensively. This indicates that an implementation with only the highly used registers implemented would reduce the register set size without much degradation of performance. Some of these 20 registers are used by the 360 for addressing purposes, but many more than the three to four registers normally used for addressing [HAM77] have low usage.

After obtaining a register reference sequence the stack algorithm is applied to determine how many register faults are generated for register set sizes less than 20. Tables 4.2, 4.3 and 4.4 are a listing for

Table 4.1(a). Register Use Data for "GAUS".

Register No.	No. of Reads	No. of Writes
1	4997	5471
2	488	479
3	1490	448
4	1488	487
5	4035	1060
6	3870	104
7	272	241
8	3266	3415
9	279	245
10	1179	836
11	276	320
12	222	283
13	0	1
14	5	1
15	0	0
16	0	15
17	1738	1757
18	13153	13155
19	3848	3242
20	3080	286
Total	43686	31846

Table 4.1(b). Register Use Data for "EIGEN".

Register No.	No. of Reads	No. of Writes
1	2388	2178
2	371	600
3	636	103
4	885	296
5	2067	432
6	1954	415
7	636	546
8	1701	1859
9	93	93
10	345	306
11	439	25
12	208	141
13	0	1
14	6	1
15	244	246
16	246	2
17	4729	3957
18	12551	11232
19	6306	4936
20	2377	1810
Total	38182	29179

Table 4.1(c). Register Use Data for "LIST".

Register No.	No. of Reads	No. of Writes
1	1594	1895
2	1220	1373
3	620	826
4	1707	1637
5	2595	2312
6	1037	1096
7	2183	1604
8	1763	1141
9	1173	1265
10	690	728
11	181	133
12	351	4
13	4	0
14	652	974
15	1188	1507
16	431	1156
17	21	21
18	21	7
19	0	0
20	0	0
Total	17431	17679

Table 4.2. Register Fault Data for "GAUS".

Register Set Size	Number of Register Faults	Register Faults per Mem. Ref.	Performance Degradation
1	22448	.2991	.5982
2	20195	.2691	.5382
3	17956	.2392	.4785
4	17447	.2325	.4649
5	14050	.1872	.3744
6	6849	.0913	.1825
7	3814	.0508	.1016
8	1863	.0248	.0496
9	1224	.0163	.0326
10	1154	.0154	.0308
11	662	.0088	.0176
12	147	.0020	.0039
13	127	.0017	.0034
14	110	.0015	.0029
15	90	.0012	.0024
16	0	.0000	.0000
17	0	.0000	.0000
18	0	.0000	.0000
19	0	.0000	.0000
20	0	.0000	.0000

Total number of memory references	75052
Total number of memory data reads	18904
Total number of memory data writes	70[illegible]
Total number of register references	75532

Table 4.3. Register Fault Data for "EIGEN".

Register Set Size	Number of Register Faults	Register Faults per Mem. Ref.	Performance Degradation
1	20098	.3188	.6375
2	14764	.2342	.4683
3	11645	.1847	.3694
4	9854	.1563	.3126
5	5745	.0911	.1822
6	3966	.0629	.1258
7	3095	.0491	.0982
8	2733	.0433	.0867
9	2253	.0357	.0715
10	1784	.0283	.0566
11	509	.0081	.0161
12	292	.0046	.0093
13	163	.0026	.0052
14	113	.0018	.0036
15	60	.0010	.0019
16	48	.0008	.0015
17	24	.0004	.0008
18	1	.0000	.0000
19	0	.0000	.0000
20	0	.0000	.0000

Total number of memory references	63048
Total number of memory data reads	16088
Total number of memory data writes	6797
Total number of register references	67361

Table 4.4. Register Fault Data for "LIST".

Register Set Size	Number of Register Faults	Register Faults per Mem. Ref.	Performance Degradation
1	9622	.1380	.2760
2	7370	.1057	.2114
3	5981	.0858	.1716
4	4841	.0694	.1389
5	3884	.0557	.1114
6	3549	.0509	.1018
7	2985	.0428	.0856
8	2535	.0364	.0727
9	1896	.0272	.0544
10	1475	.0212	.0423
11	1115	.0160	.0320
12	593	.0085	.0170
13	280	.0040	.0080
14	43	.0006	.0012
15	13	.0002	.0004
16	13	.0002	.0004
17	0	.0000	.0000
18	0	.0000	.0000
19	0	.0000	.0000
20	0	.0000	.0000

Total number of memory references	69723
Total number of memory data reads	11640
Total number of memory data writes	8451
Total number of register references	35110

GAUS, EIGEN and LIST respectively, of the total number of register faults generated, the number of register faults per memory reference and the performance degradation for register set sizes from 1 to 20.

The interesting figure here is the low number of register faults per memory reference even with only one data register. The highest being only slightly more than .3 with a performance degradation of .64. The two numerical analysis programs, G. S and EIGEN, have fairly similar distributions but the file manipulation program, LIST, has the fewest register faults until the register set size reaches seven.

Figure 4.1 is a graph of the data from Tables 4.2 through 4.4. This graph shows the trend more clearly. As the register set increases, the register faults decrease in a manner roughly approximating an exponential decay. All three programs have a similar number of register faults when the register set is greater than seven. Both numerical analysis programs have a jog in the graph for register set sizes from four to six. The file manipulation program is rather smooth.

Since three to four registers on the average are used for addressing, no more than 17 registers are usually available for data. This is why the number of register faults is very close to zero for register set sizes of about 15 to 16.

4.6 Evaluation of Performance

Figure 4.1 gives effective execution time as a function of the number of data registers for a single process CPU. The worst degradation of performance is when R is one. Figure 4.1 indicates that T(1) is about 1.6 for the two numerical analysis programs (EIGEN and GAUS) and about 1.3

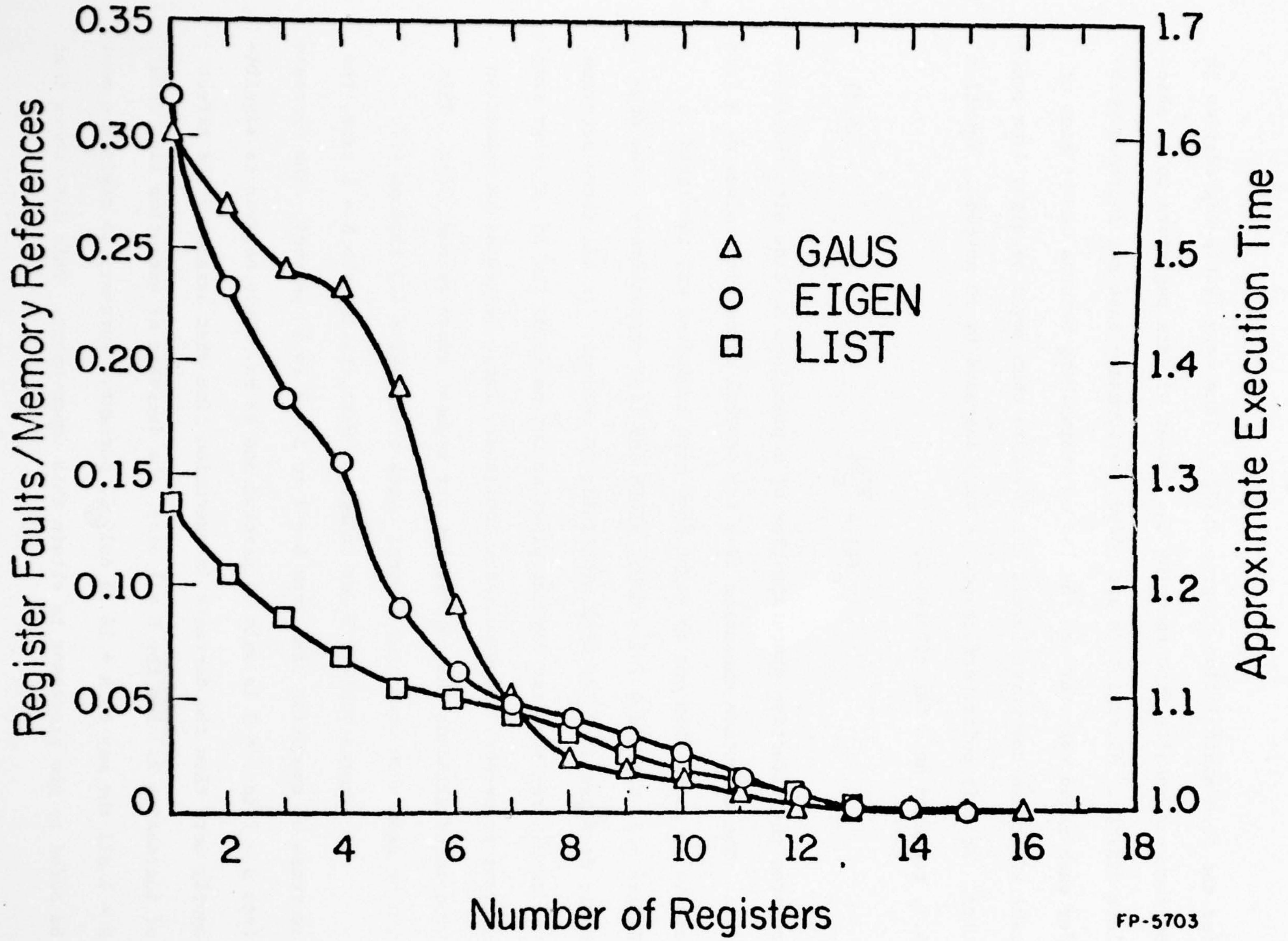


Figure 4.1. Plot of Register Faults/Memory Reference vs. Number of Data Registers.

for the file manipulation program (LIST). Thus very little degradation in performance results from reducing the number of data registers to an absolute minimum. As registers are added, performance does not increase much for each added register and the law of diminishing returns can be seen to take effect as the curve levels out for more than seven or eight data registers. At this point performance is only degraded by 10 percent. Equation 4.3, repeated here for reference,

$$T_e(R) = \frac{T(R)}{S} \qquad (4.3)$$

relates the effective execution time of a pipelined machine with the above data. The effective execution time for several cases with values of R from one to 16 and S from one to eight have been evaluated and tabulated in Tables 4.5, 4.6 and 4.7 for GAUS, EIGEN and LIST respectively. The dramatic decrease in $T_e(R)$ with pipelining is evident. In all three programs, the single register two segment pipeline out performs the 16 register single segment processor. As pipelining continues, large decreases in execution time occur although the improvement per segment added becomes less. This can be seen when comparing several cases from Tables 4.5 through 4.7.

Examine Table 4.5 for example. Here, taking the R = 1 case, the decrease of execution time from S = 1 to S = 2 is 50 percent. The decrease from S = 2 to S = 3 is only 33 percent and so on. This, however, is significantly more than the decrease in execution time when examining the effect of increasing R. For the S = 1 case, the decrease of execution time from R = 1 all the way to R = 16 is only 37 percent. Further, 15 registers must be added to the processor to obtain this improvement. This data shows that

Table 4.5. Effective Execution Time for Various Processors for the Program "GAUS"

R \ S	1	2	3	4	5	6	7	8
1	1.598	.799	.533	.400	.320	.266	.228	.200
2	1.538	.769	.513	.385	.308	.256	.220	.192
3	1.478	.739	.493	.370	.296	.246	.211	.185
4	1.465	.733	.488	.366	.293	.244	.209	.183
5	1.374	.687	.458	.344	.275	.229	.196	.172
6	1.183	.592	.394	.296	.237	.197	.169	.148
7	1.1016	.551	.367	.275	.220	.184	.157	.138
8	1.0497	.525	.350	.262	.210	.175	.150	.131
9	1.0326	.516	.344	.258	.207	.172	.148	.129
10	1.0308	.515	.344	.258	.206	.172	.147	.129
11	1.0176	.509	.339	.254	.204	.170	.145	.127
12	1.0039	.502	.335	.251	.201	.167	.143	.125
13	1.0034	.502	.334	.251	.201	.167	.143	.125
14	1.0029	.501	.334	.251	.201	.167	.143	.125
15	1.0024	.501	.334	.251	.200	.167	.143	.125
16	1.0000	.500	.333	.250	.200	.167	.143	.125

Table 4.6. Effective Execution Time for Various Processors for the Program "EIGEN"

R \ S	1	2	3	4	5	6	7	8
1	1.638	.819	.546	.410	.328	.273	.234	.205
2	1.468	.734	.489	.367	.294	.245	.210	.184
3	1.369	.685	.456	.342	.274	.228	.196	.171
4	1.313	.657	.438	.328	.263	.219	.188	.164
5	1.182	.591	.394	.296	.236	.197	.169	.148
6	1.126	.563	.375	.282	.225	.188	.161	.141
7	1.098	.549	.366	.275	.220	.183	.157	.137
8	1.087	.544	.362	.272	.217	.181	.155	.136
9	1.071	.536	.357	.268	.214	.179	.153	.134
10	1.057	.529	.352	.264	.211	.176	.151	.132
11	1.016	.508	.339	.254	.203	.169	.145	.127
12	1.009	.505	.336	.252	.202	.168	.144	.126
13	1.005	.503	.335	.251	.201	.168	.144	.126
14	1.004	.502	.335	.251	.201	.167	.143	.126
15	1.002	.501	.334	.251	.200	.167	.143	.125
16	1.002	.501	.334	.251	.200	.167	.143	.125

Table 4.7. Effective Execution Time for Various Processors for the Program "LIST"

R \ S	1	2	3	4	5	6	7	8
1	1.276	.638	.425	.319	.255	.213	.182	.160
2	1.211	.605	.404	.303	.242	.202	.173	.151
3	1.172	.586	.391	.293	.234	.195	.167	.147
4	1.139	.569	.379	.285	.228	.190	.163	.142
5	1.111	.555	.370	.278	.222	.185	.159	.139
6	1.102	.551	.367	.275	.220	.184	.157	.138
7	1.086	.543	.362	.271	.217	.181	.155	.136
8	1.073	.537	.358	.268	.215	.179	.153	.134
9	1.054	.527	.351	.264	.211	.176	.151	.132
10	1.042	.521	.347	.261	.208	.174	.149	.130
11	1.032	.516	.344	.258	.206	.172	.147	.129
12	1.017	.509	.339	.254	.203	.170	.145	.127
13	1.008	.504	.336	.252	.202	.168	.144	.126
14	1.001	.501	.334	.250	.200	.167	.143	.125
15	1.000	.500	.333	.250	.200	.167	.143	.125
16	1.000	.500	.333	.250	.200	.167	.143	.125

AD-A056 110 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
ARCHITECTURE FOR MULTIPLE INSTRUCTION STREAM LSI PROCESSORS.(U)
OCT 77 W J KAMINSKY DAAB07-72-C-0259
UNCLASSIFIED R-796 NL

2 OF 2
ADA
056110

END
DATE
FILMED
8-78
DDC

adding registers to decrease execution time with a memory limited system, as described here, is not very effective. Furthermore, if a system were not memory limited, or if one of our other assumptions did not hold, pipelining would be preferred over added registers to an even greater extent.

Examining the tables for the other program traces gives similar results. Adding registers for the case of the LIST program is even less effective than for the other two programs. This is probably due to less locality of data in this program and the use of fewer temporary results.

The tradeoff between adding registers vs pipelining can be observed by examining graphs of constant performance as a function of R and S. Figures 4.2, 4.3 and 4.4 are plots of constant performance, for the programs GAUS, EIGEN and LIST respectively, which are computed from data in Tables 4.5 through 4.7. These curves represent "equiperformance" profiles and are labeled with their relative performance with respect to a single stream, 16 data register processor. The x's are plotted for integer values of S and interpolated for R to obtain smooth curves which indicate the lower bound for values of R and S of systems for each level of performance plotted.

As one follows a curve of constant performance from the one register case toward increasing R, the curve "flattens" rapidly toward an asymptote equal to $S = 1/T_e(16)$. However, even when R is small, the slope is not great and the increase necessary in S to maintain constant performance is small.

The flatter the curve, the less effective the addition of registers is. As the effective execution time decreases (performance increases) the curves become more steep for low values of R. This indicates that when

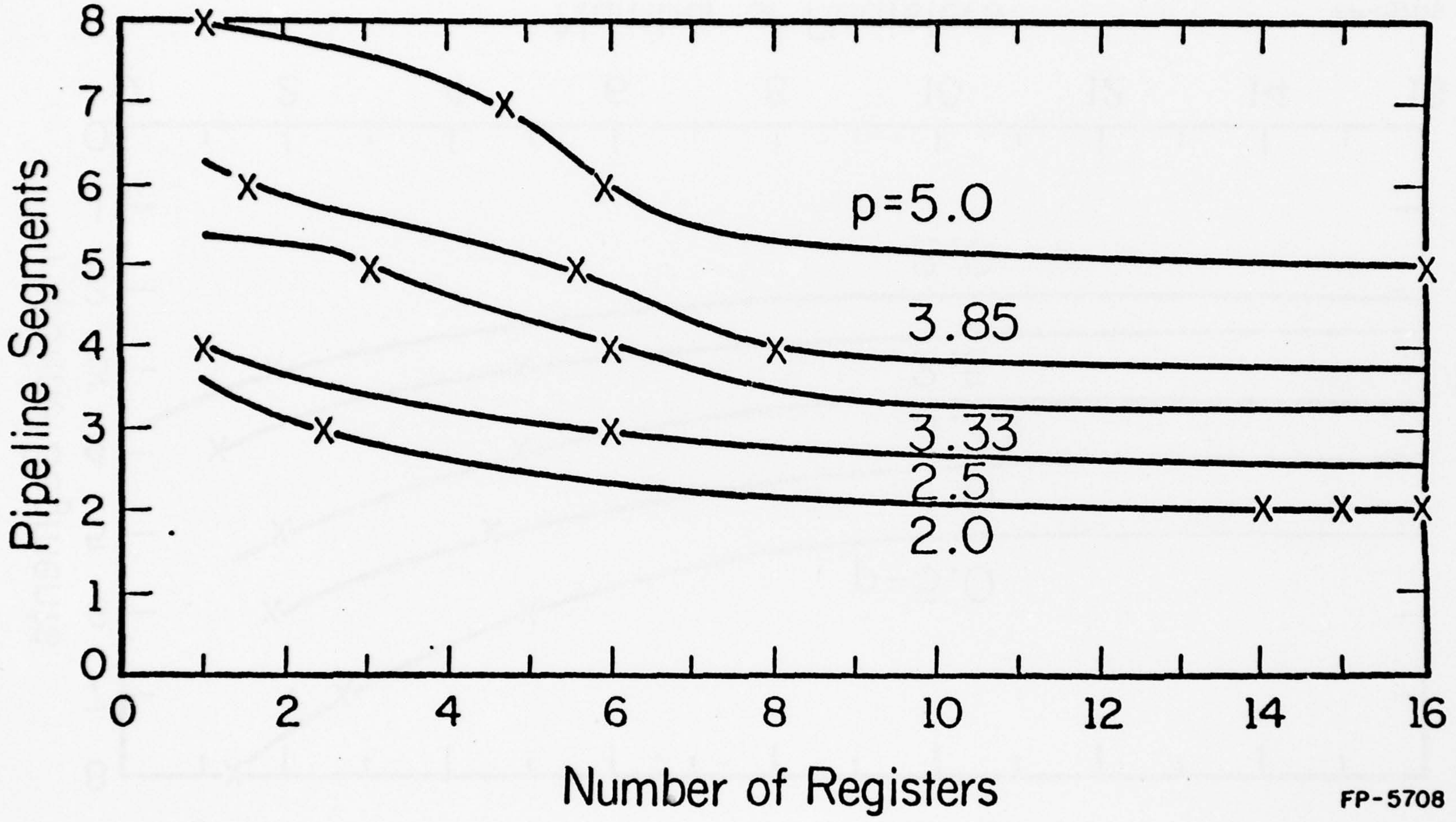


Figure 4.2. Constant Performance Plot for "GAUS."

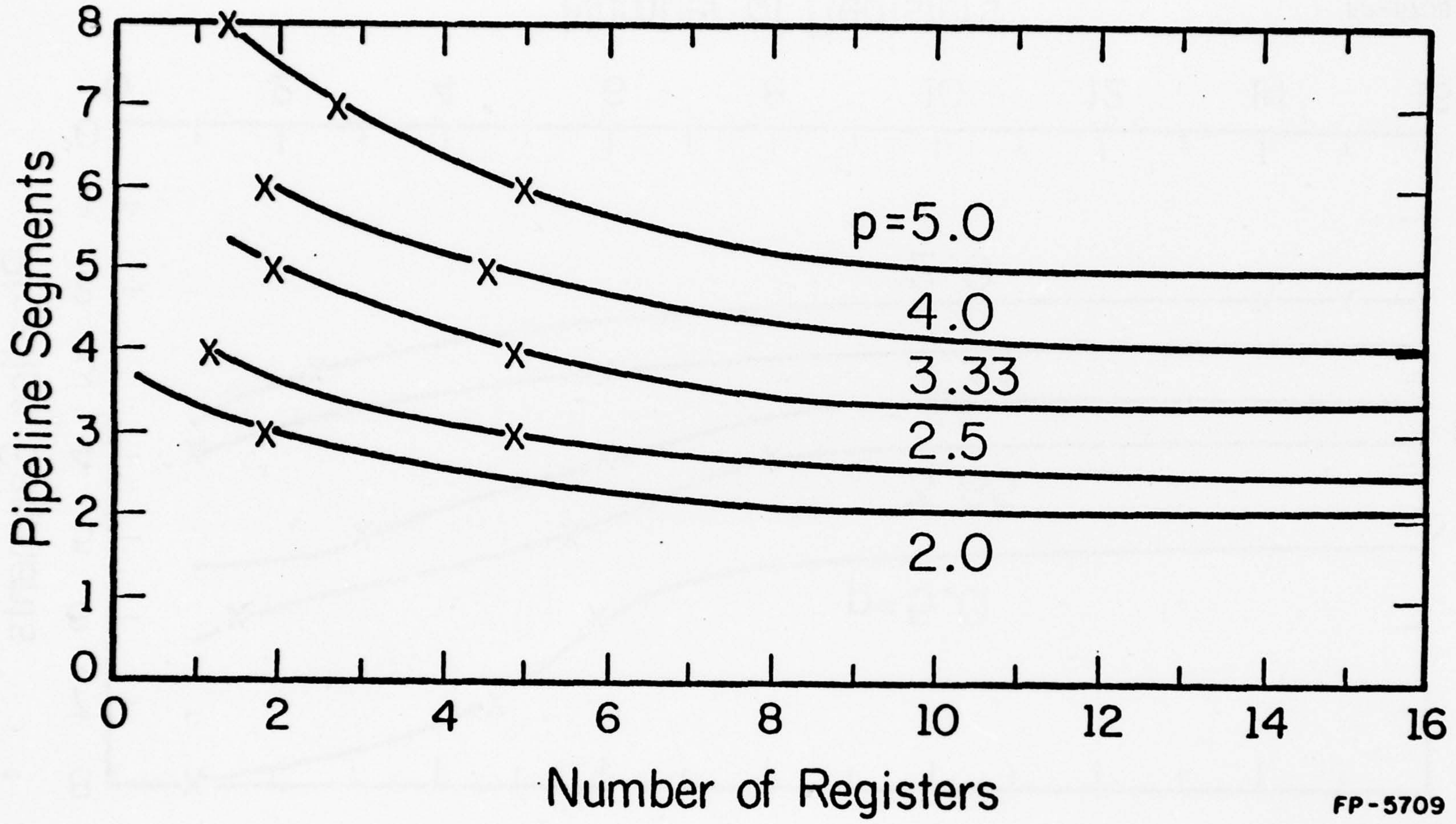


Figure 4.3. Constant Performance Plot for "EIGEN."

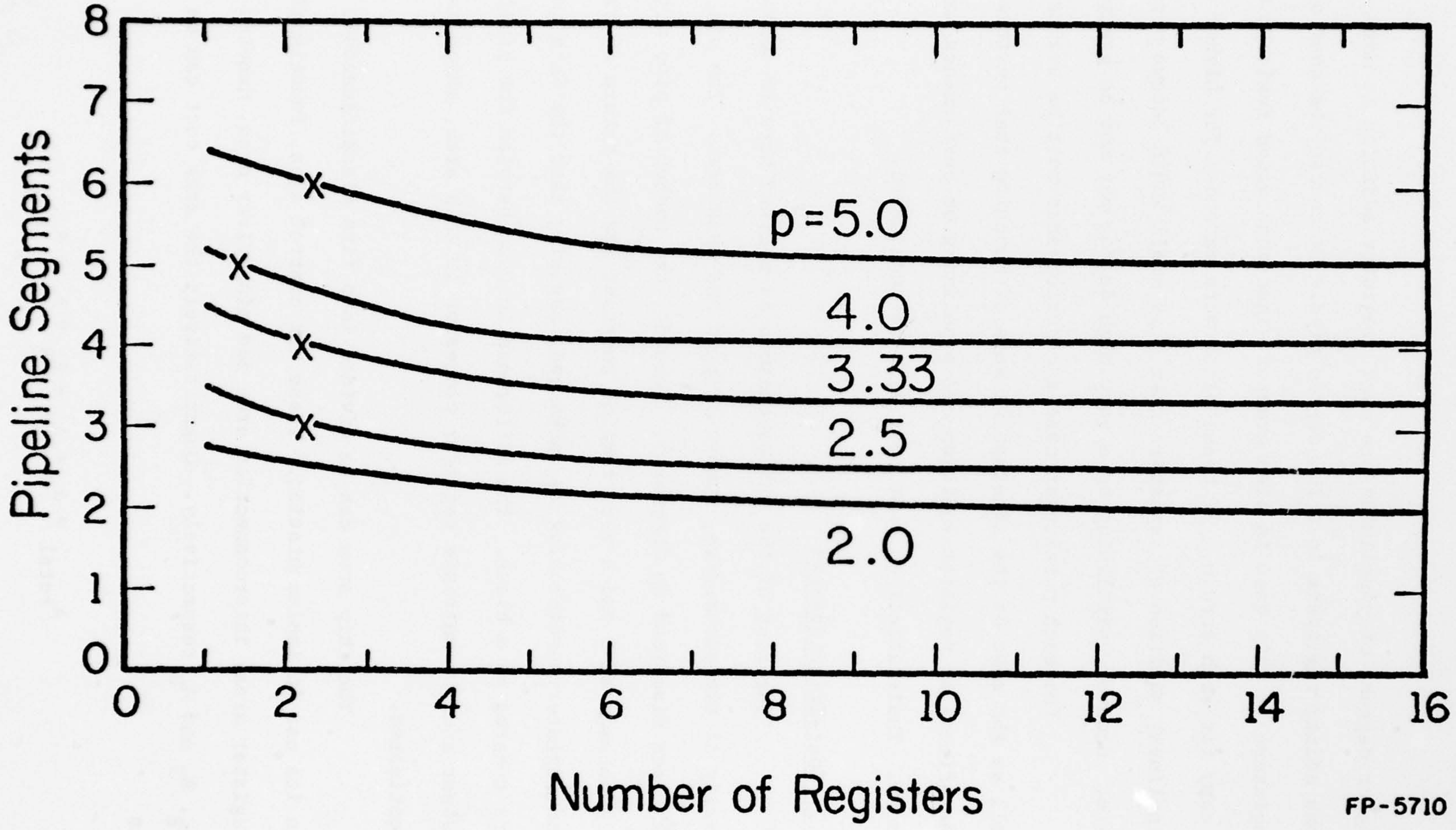


Figure 4.4. Constant Performance Plot for "LIST."

higher degrees of pipelining are used, register addition is more effective than adding registers in a low degree pipe. Of course the cost of adding registers in this case is also greater since each added register must have a copy for each stream. If these plots were extended for higher degree pipelines, addition of registers when R is small would become more effective. However, pipelining to a very high degree may not be practical.

For each particular case the performance must be evaluated as well as the cost of the alternative ways of reaching that performance. The effects of register addition and pipelining on performance have been shown. Their effect on cost must also be considered.

4.7 Cost-Effectiveness

The cost of the LSI processors as modeled here is composed primarily of two parameters, number of pins and chip area. The pin problem has been discussed in Chapter 3. However, the number of pins required by a processor as S and R vary remains constant and the nature of bit slice and register partitioning is unchanged, assuming that the data registers are treated as a block. The difference in cost between the pipelined processor and the multiple register processor is chip area, which is now investigated.

The chip area can be divided into five classifications according to its use in implementation. These are control area, functional unit area, register area, interconnection area and pin driver area, denoted A_C, A_F, A_R, A_I and A_D respectively. Quantitatively the area cost can be expressed as

$$A_{total} = A_C + A_F + A_R + A_I + A_D. \qquad (4.5)$$

These area classifications can be grouped into two groups, those that are constant for the range of processors discussed in the chapter and those which are dependent on degree of pipelining and/or number of data registers. Since the pin requirements over the range of processors do not change, A_D is constant with all implementations studied. All processors execute the same instruction set and control is approximately the same other than that needed to control the additional functional units in the pipelined case and additional registers in the multiple register case. Control can be grouped as part constant and part incremental dependent on the values of R and S. Functional unit area has a minimum component representing a minimal set of logic units and a component to represent the additional units needed when pipelining. The register area can be represented with four components, a component for registers other than data registers, a component for data registers and two components representing copies of both data and nondata registers needed for pipelining. The interconnection area is needed for all control and data signals and can be attributed to the functional units and registers that require these signals. The interconnection cost is therefore not treated separately, but rather absorbed into the cost of appropriate functional units and registers.

The total area cost thus can be expressed approximately as

$$C = k_0 + k_1 R + k_2(S-1) + k_3 R(S-1) \tag{4.6}$$

where k_0 represents the constant cost of driver area, constant portion of control and functional units, the raw data register cost and the interconnection area associated with them. k_1 represents the area cost of each data register for the nonpipelined case and the associated control and interconnection area for these registers. k_2 represents the additional area cost required for pipelining the nondata registers and the additional logic unit area needed as pipelining is increased. k_3 represents the additional area for registers when pipelining the data registers. Both k_1 and k_3 depend on (S-1) instead of S since these are marginal costs beyond those of the basic single process CPU.

The major cost differences over the range of processors are additional functional unit cost for the pipelined case, the additional register cost for either the multiple register or pipelined case and the interconnection cost associated with these units. The multiple register processor does not require any additional functional units over those of the basic CPU except for multiplexers and demultiplexers needed by the additional registers. Most of the nonconstant cost for this case represents additional register area and interconnection area. This is what k_1 represents.

The pipelined processor adds area for replication of each register, data and nondata, needed for each segment of the pipe, and for the additional functional units required when one is used more than once during a cycle, provided the segmentation of the pipe divides the processor cycle between separate uses of the functional unit during a cycle. There is some area saved, however, with pipelining. When an additional functional unit is added, fewer interconnections are needed since the units can be placed closer to the sections of the processor requiring their use. The other savings of area is that the interconnection area needed per register to connect registers in the pipelined fashion is likely to be less than the interconnection area required per register to extend the basic register set. This is because the pipelined structure of layout is very regular and regular patterns tend to require less interconnection area than less well-structured patterns do. Since, in the pipelined processor, data is always shifted to the next register, dynamic registers can be used throughout. These require less area than do static registers.

The relative costs of the various uses of area vary widely with the specifics of the CPU to be implemented and the technology used. With additional information, values for k_0, k_1, k_2 and k_3 can be assigned and cost tradeoffs evaluated.

To obtain a crude approximation of the relative costs of alternative processors, certain simplifying assumptions can be made for the constants in Equation 4.6. Assuming that each register will cost the same amount of area regardless of its use as a new register or a replication required for pipelining an old register, coefficient k_1 will be set equal to k_3. Secondly, it is assumed that the cost of the additional functional units for pipelining is balanced approximately by the savings in area for interconnection to these units and the reduced interconnection area per register required when pipelining. Utilizing this and the first assumption, the constant k_2 will be set equal to the number of nondata registers times k_1. For the processor of Figure 2.5, $k_2 = 5k_1$.

Equation 4-6 can be rearranged as

$$C = k_0 - k_2 + (k_1 - k_3)R + k_2 S + k_3 RS. \tag{4.7}$$

Invoking the previous assumptions and substituting $k_1 = k_3$ and $k_2 = 5k_1$ gives

$$C = k_0 - 5k_1 + k_1(5+R)S \tag{4.8}$$

Normalizing Equation 4.8 to the constant term $k_0 - 5k_1$ and letting $C'_{R,S} = \frac{C}{k_0-5k_1}$ and $K' = \frac{k_1}{k_0-5k_1}$ gives

$$C'_{R,S} = 1 + K'(5+R)S \tag{4.9}$$

Using this equation, the approximate costs for various processors with performance given in Tables 4.5 through 4.7 can be compared.

It is interesting to compare the cost of the single process, many register CPU (R=16, S=1) to that of the single register, two segment CPU (R=1, S=2). For the program GAUS, the effective execution time for R = 1, S = 2 is about .8, which is better than the single process CPU for R = 16, for which $T_e(16) \simeq 1.0$. The cost for the single process CPU (R=16, S=1) is

$$C'_{16,1} = 1 + K'(5+16)1 = 1 + 21K' \tag{4.10}$$

For the two segment pipelined CPU (R=1, S=2)

$$C'_{1,2} = 1 + K'(5+1)2 = 1 + 12K'. \tag{4.11}$$

Thus, under the assumptions made about costs, the best single process CPU costs significantly, $9K'$, more (provided K' is significant compared to 1) than the worst performing pipelined CPU and that pipelined CPU performs 25 percent better. Thus the R,S = 1,2 processor is estimated to be both cheaper and faster than the R,S = 16,1 processor.

Several other cases, comparing the costs of single data register processors with various levels of pipelining and multiple register processors with less pipelining but equal performance are summarized in Table 4.8 for the program GAUS. In each case, for equal performance, the higher level pipeline was cheaper. Thus, based on the cost assumptions, it is better to increase the level of pipelining than to add data registers. Comparisons using the programs "EIGEN" and "LIST" are summarized in Tables 4.9 and 4.10 respectively. Similar results are attained for "EIGEN" and even more convincing results for "LIST."

These cost results are only crude approximations, but if actual parameters could be obtained for k_1, k_2 and k_3 results could be obtained for actual processors in a given technology. It is hard to imagine that more precise modeling and actual parameters for a given technology would alter the conclusions drawn here. The intent here has been only to approximate the nature of cost and performance dependency of pipelining vs the use of additional data registers.

4.8 Conclusion

It has been shown in this chapter that performance is enhanced more by pipelining than by adding a significant number of data registers. In fact, it was found that, when extending the data register set of a single process CPU even to 16 data registers, it performed no better than 80 percent of the performance of a single data register, two segment pipelined CPU.

These performance evaluation were based on the data register reference pattern within the IBM 360 for three program traces. Although the 360's performance is much different than that of a small processor, these differences by and large detract from the value of pipelining relative

Table 4.8. Costs for Processors of Equal Performance for the Program "GAUS"

P(R)	R	S	C'
1.89	8	2	$1 + 26K'$
	1	3	$1 + 18K'$
2.50	6	3	$1 + 33K'$
	1	4	$1 + 24K'$
3.77	8	4	$1 + 52K'$
	1	6	$1 + 36K'$
5.00	6	6	$1 + 66K'$
	1	8	$1 + 48K'$

Table 4.9. Costs for Processors of Equal Performance for the Program "EIGEN"

P(R)	R	S	C'
1.8	8	2	$1 + 26K'$
	1	3	$1 + 18K'$
3.0	4	4	$1 + 36K'$
	1	5	$1 + 30K'$
3.7	3	5	$1 + 40K'$
	1	6	$1 + 36K'$
5.4	7	6	$1 + 78K'$
	2	8	$1 + 56K'$

Table 4.10. Costs for Processors of Equal Performance for the Program "LIST"

P(R)	R	S	C'
3.9	12	4	$1 + 68K'$
	1	5	$1 + 30K'$
4.7	8	5	$1 + 65K'$
	1	6	$1 + 36K'$
5.5	7	6	$1 + 72K'$
	1	7	$1 + 42K'$
6.2	5	7	$1 + 70K'$
	1	8	$1 + 48K'$

to using more registers for a single stream processor. The conclusions reached would be even more strongly justified by more realistic data.

The number of memory references was used as the basis for performance for a single processor as register set size varied. The various performance levels as a function of the level of pipelining and the register set size were evaluated from the memory reference data and summarized in Tables 4.5 through 4.7. An approximate cost evaluation was performed on this data which supported the previous conclusions on grounds of cost as well as performance. More specific performance/cost data is dependent on the specific machine characteristics and capabilities and on the technology used, but for the data gathered here, pipelining is an efficient and practical architecture for an LSI processor.

CHAPTER 5. CONCLUSION

The purpose of this research is to determine future architectures suitable for implementation with LSI, taking into consideration the trend in growth of practical LSI chip capacity. Since the area of the chip has been and is expected to continue increasing exponentially, it was determined that pin limitations of future LSI implementations would be a major constraint on feasible implementations. In order to obtain high pin utilization, which reduces the number of pins required on the chip, as well as high performance, a multiprocess pipeline was chosen as being most suited to achieve these objectives. It needed to be shown, however, how such a processor could be organized. A methodology to determine the organization and final layout of a multiprocess pipelined CPU suitable for LSI implementation was developed. This methodology is based on a one memory port processor, which is necessary to minimize the number of pins required. The methodology begins with a given instruction set, designed for efficient execution of the intended program mix.

The objective of the organization is to generate memory references as soon as possible during the processor cycle while minimizing the number of internal resources. By using the time between memory references of one process, to issue memory references from other processes, high throughput can be attained with only one memory port. This organization is particularly suited for LSI implementation because of its regular structure which reduces the amount of interconnection area required to allow the addition of active devices.

It is also desirable to know what effects would occur if a processor had to be partitioned into more than one chip and how this could effectively be done. Two reasonable methods of partitioning were found, the bit slice and register partition. These were analyzed to determine how the pin count was affected by partitioning. It was found that the bit slice partition is useful to partition a processor with a large data word, compared to the number of control lines. The register partition performs better when the number of control lines necessary is large compared to the data word size and when cut sets exist which divide the I/O buses and chip area appropriately without cutting too many internal data paths. A combination of the two partitioning methods is best when a high degree of partitioning is required. In all cases, however, the number of pins for the chip set becomes very high as partitioning is increased. It was also found that the effectiveness of each partitioning method in reducing the number of pins per chip decreases as partitioning increases.

The performance of the pipelined multiprocessor was compared with that of a single process CPU with multiple data registers. A multiple data register processor is one approach some manufacturers have used to utilize additional area to improve performance without increasing pin requirements. The pipelined multiprocessor uses few data registers to allow area for adding registers required for pipelining. To determine the effectiveness of adding additional registers, the register usage pattern of a processor with many data registers, the IBM System 360, was studied. Traces of several programs were analyzed to determine the degradation in performance as the register set was reduced. This data was then used to determine the performance of various single data register pipelined processors and several single process CPU's with multiple data registers. It was found that pipelining was substantially more effective in increasing the performance than adding data registers. The cost of these two types of processors were modeled and an approximate cost comparison made for processors with the same performance. It was found that the pipelined processor was more cost-effective. These results show that the pipelined multiprocessor is a good organization for future LSI designs.

There are several other aspects of this processor which could be investigated further. The data register effectiveness has been analyzed but it would be interesting to analyze the effect of the number and type of addressing registers on performance. The program counter register is probably necessary but the effectiveness of other addressing registers such as the stack pointer register, indexing registers and base registers needs to be investigated.

Also of importance to the pipelined multiprocessor is the selection of an instruction set. Sequences of basic instructions could also be analyzed to find optimal instruction sets for pipelined processors. The layout of this processor is dependent, specifically on the type of instructions chosen to be executed by the processor. Careful selection could improve the performance and retain a desirable organization for the processor. It may also be determined that certain job requirements often encountered (e.g., execution of parallel real time tasks) can be effectively handled with a pipelined multiprocessor.

The method of communicating between processes and scheduling of processes for a pipelined processor should also be studied. Certain environments such as time sharing, lend themselves very well toward a pipelined multiprocessor, but system utilization in different environments needs to be analyzed. There are operating systems which control multiple processes now and their adaptation to this pipelined multiprocessor should be straightforward.

BIBLIOGRAPHY

BEL66	Belady, L. A., "A Study of Replacement Algorithms for a Virtual Storage Computer," IBM Sys. J., Vol. 9, 1966, 78-101.
BLA77	Blake, R. P., "Exploring a Stack Architecture," Computer, Vol. 1, No. 5, May 1977, 30-38.
BRI77	Briggs, F. A., "Memory Organization and Their Effectiveness for Multiprocessing Computers," CSL Report R-768, University of Illinois, May 1977.
DEC77	DEC (Digital Equipment Corporation), Processor Handbook PDP11/45, 1974.
FOR62	Ford, L. R. and Fulkerson, D. R., Flows in Networks, Princeton University Press, 1962.
HAM77	Hammerstrom, D. W., "Analysis of Memory Addressing Architecture," CSL Report R-777, University of Illinois, July 1977.
HP	HP (Hewlett-Packard Co.), A Pocket Guide to HP Computers (HP 2114A, 2115A and 2116B Computers).
KET76	Ketelsen, M. L., "An Integrated Circuit Fault Model for Digital Systems," CSL Report R-743, University of Illinois, September 1976.
KOG77	Kogge, P. E., "The Microprogramming of Pipelined Processors," Proceedings of the Fourth Annual Symposium on Computer Architecture, April 1977, 63-69.
LUN77	Lunde, A., "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," Communications of the ACM, Vol. 20, No. 3, March 1977, 143-153.
LUN74	Lunde, A., "Evaluation of Instruction Set Processor Architecture by Program Tracing," Ph.D. Thesis, Carnegie-Mellon U., Pittsburgh, PA, July 1974.
MAY72	Mayeda, W., Graph Theory, John Wiley and Sons, Inc., 1972, 420-443.
NOY76	Noyce, R. N., "From Relays to MPU's," Computer, Vol. 9, No. 12, Dec. 1976, 26-29.
PEN72	Penney, W. M. and Law, L., ed., MOS Integrated Circuits, Van Nostrand Reinhold Co., 1972.
SHA74	Shar, L. E. and Davidson, E. S., "A Multiminiprocessor System Implemented Through Pipelining," Computer, Feb. 1974, 42-51.

APPENDIX A. INSTRUCTION PRECEDENCE GRAPHS

The following is a list of the instruction precedence graphs for each instruction of the instruction set listed in Table 2.7. Table A.1 describes each of the functional components used in the precedence graphs.

Table A.1. Functional Components

A1	$MD \rightarrow A$
A2	MD ADD or XOR or SUB or AND or IOR $A \rightarrow A$
A3	A SHIFT $\rightarrow A$
A4	TEST A
A5	$\overline{A} \rightarrow A$
A6	$\overline{A} + 1 \rightarrow A$
A7	$A\ (\pm)\ 1 \rightarrow A$
B1	$MD \rightarrow MA$
B2	$PC \rightarrow MA$
B3	$MD + MA \rightarrow MA$
B4	$SP \rightarrow MA$
D1	$MEM_{DATA} \rightarrow MD$
D2	$PC \rightarrow MD$
D3	$A \rightarrow MD$
I	$MEM_{DATA} \rightarrow IR$
M1	$MA \rightarrow MEM_{ADDR}$
M2	$MD \rightarrow MEM_{DATA}$
P1	$PC + 1 \rightarrow PC$
P2	$MA \rightarrow PC$
S1	$SP + 1 \rightarrow SP$
S2	$MA \rightarrow SP$

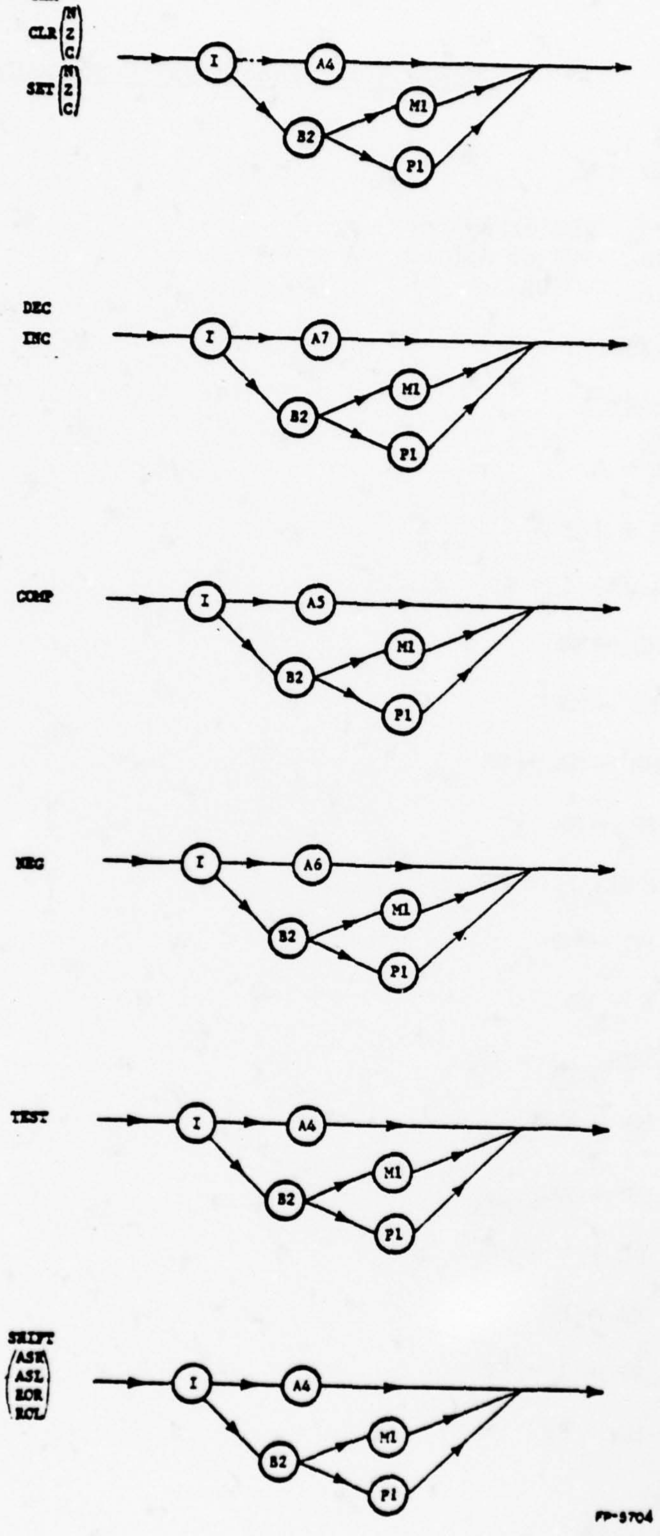


Figure A.1. Instruction Precedence Graphs I.

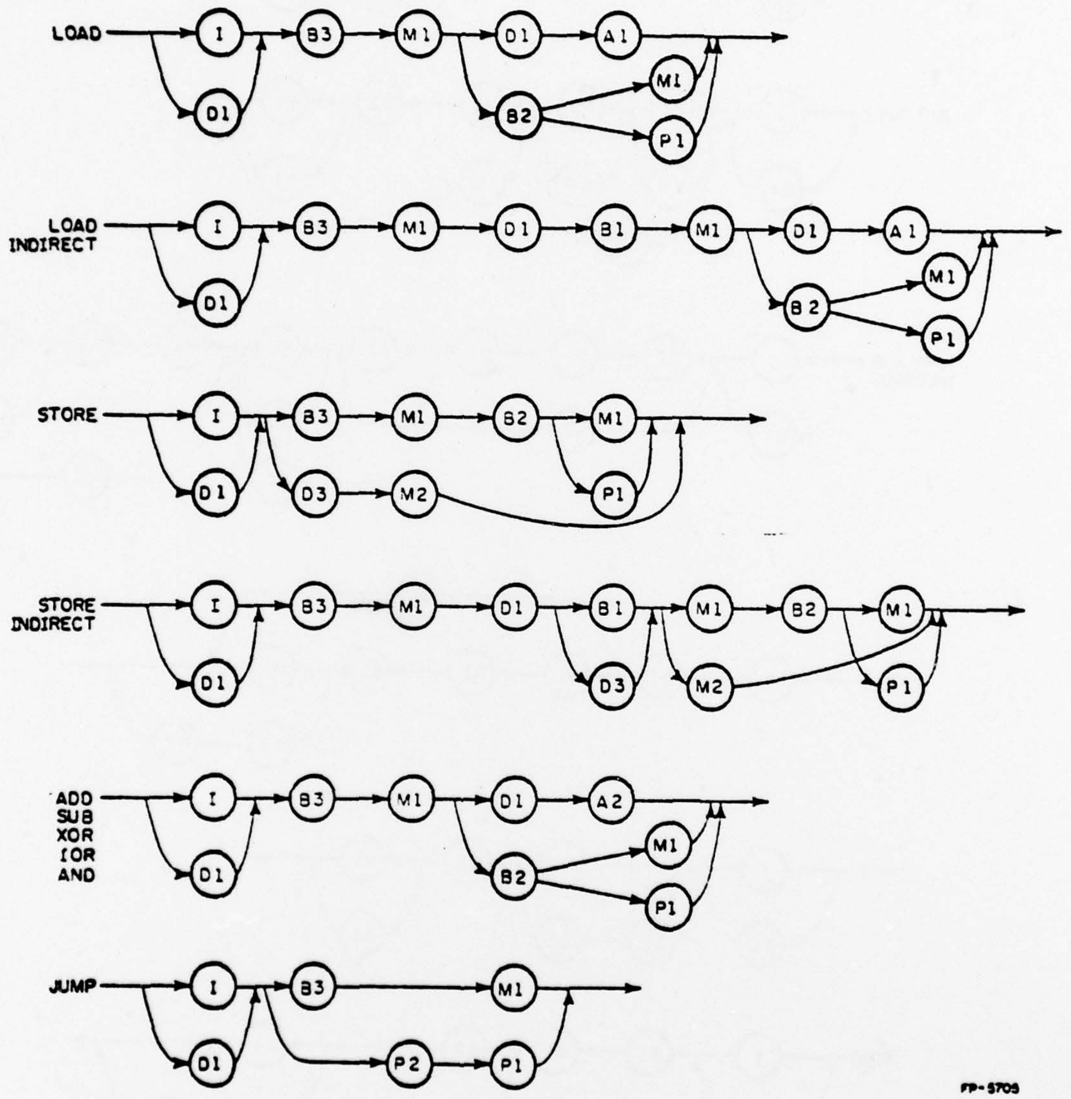


Figure A.2. Instruction Precedence Graphs II.

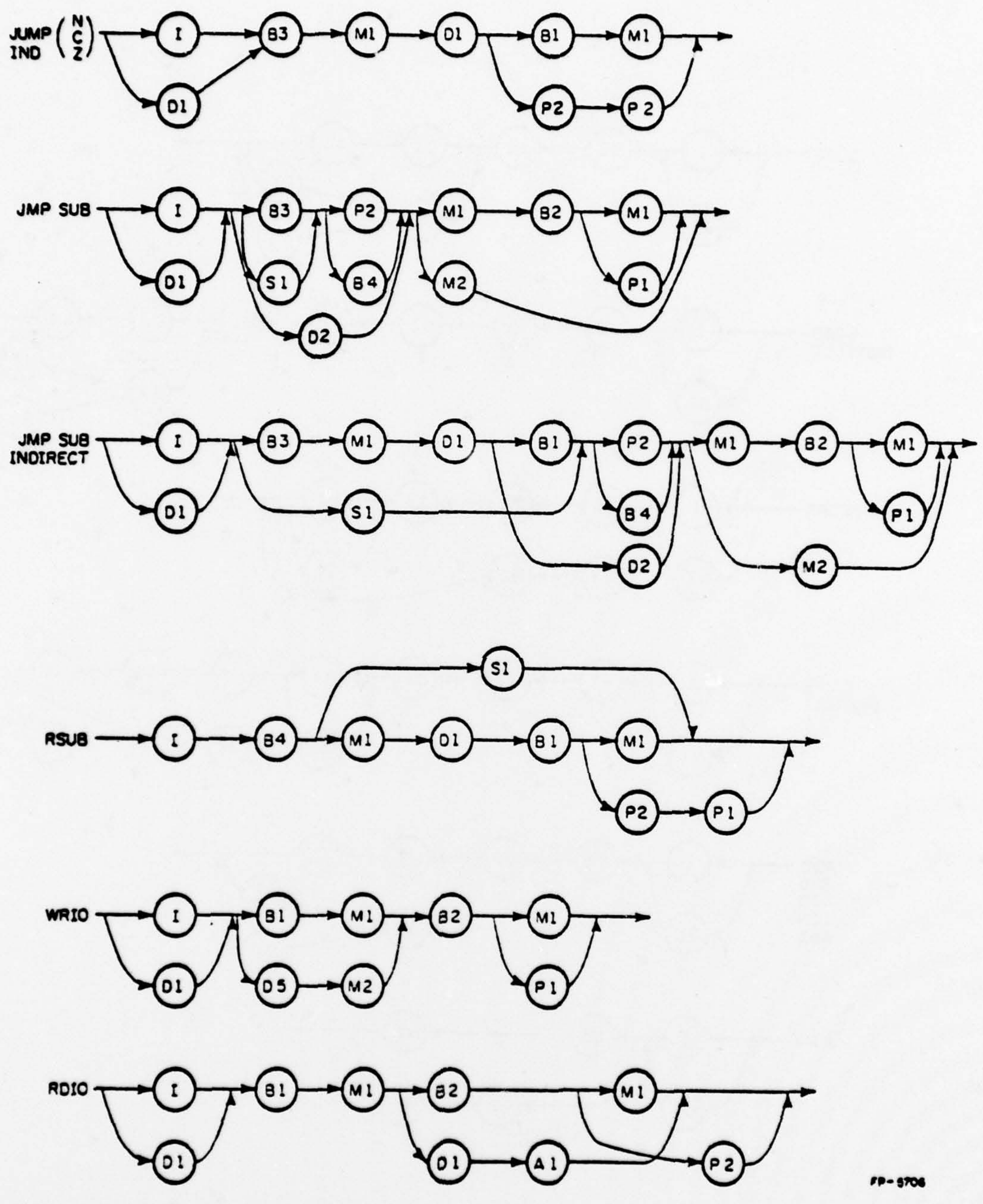


Figure A.3. Instruction Precedence Graphs III.

APPENDIX B. USING GRAPHS TO FIND REGISTER PARTITIONS

In order to perform a register partition on a processor, the interconnection structure of the processor must be studied to find the best places to divide the processor. This appendix describes a heuristic using a graphical representation of the processor to find desirable partitions. This problem may appear to resemble the well-known minimal cut problem, but differs since a cut between two particular vertices is not required. Assignment of vertices is dependent on the interconnection pattern between vertices and the amount of area required by each vertex such that a near equal division of area is obtained. The following procedure can be used to obtain reasonable partitions.

Figure B1 is a graph derived in Chapter 3 for the computer used as an example in this thesis. The first step is to determine the connection and terminal capacity matrices for this graph. Each vertex in this graph is identified both by the register or communication path it represents and a unique number which will be used to identify it in the matrices (the communication path vertices are only identified with a number). The connection matrix is determined easily by inspection and is shown in Equation B.1 for the example.

The connection matrix defines all connections between vertices. In order to find partitions the terminal capacity matrix [May 72] is first used. This matrix specifies, for any graph, the maximum flow between pairs of vertices which is equal to the minimum number of connections to be cut

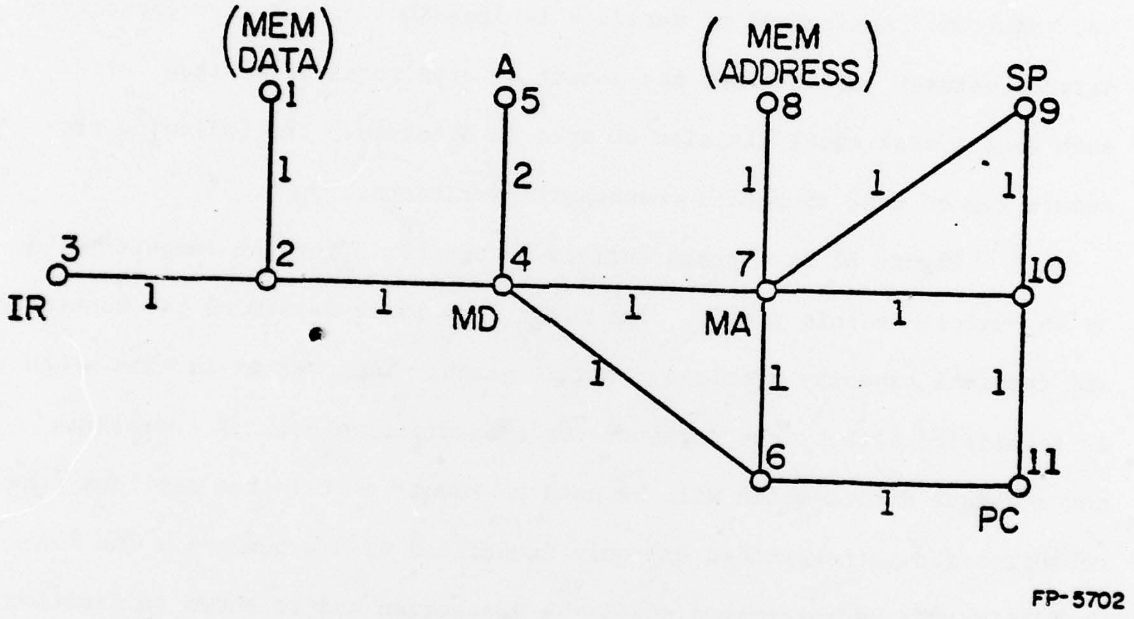


Figure B.1. Graphical Model of Processor.

$$C_1 = \begin{array}{c|ccccccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
1 & d & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & d & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & d & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 1 & 0 & d & 2 & 1 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 2 & d & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 1 & 0 & d & 1 & 0 & 0 & 0 & 1 \\
7 & 0 & 0 & 0 & 1 & 0 & 1 & d & 1 & 1 & 1 & 0 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & d & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & d & 1 & 0 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & d & 1 \\
11 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & d
\end{array} \qquad (B.1)$$

if these vertices are divided [FOR 62]. The terminal capacity matrix for the example is shown in Equation B.2.

$$T_1 = \begin{array}{c|ccccccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
1 & d & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
2 & 1 & d & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
3 & 1 & 1 & d & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
4 & 1 & 1 & 1 & d & 2 & 2 & 2 & 1 & 2 & 2 & 2 \\
5 & 1 & 1 & 1 & 2 & d & 2 & 2 & 1 & 2 & 2 & 2 \\
6 & 1 & 1 & 1 & 2 & 2 & d & 3 & 1 & 2 & 3 & 2 \\
7 & 1 & 1 & 1 & 2 & 2 & 3 & d & 1 & 2 & 3 & 2 \\
8 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & d & 1 & 1 & 1 \\
9 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 1 & d & 2 & 2 \\
10 & 1 & 1 & 1 & 2 & 2 & 3 & 3 & 1 & 2 & d & 2 \\
11 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 1 & 2 & 2 & d
\end{array} \qquad (B.2)$$

The terminal capacity matrix indicates vertex pairs which have a large number of connection paths. (The number of connection paths is equal to the maximum number of signals that could start at one vertex and reach the other using disjoint paths in the graph. These connection paths are useful for determining groups which are highly interconnected and should be placed on the same chip.

Step two groups vertices in the matrix with a large number of connection paths between them and places those vertices which are defined to be external connections. This grouping is performed by interchanging pairs of rows and columns in the matrix. The i^{th} group is denoted as g_i. First the outside connection vertices are grouped together. In the example, vertices 1 and 8 representing the memory data and address busses, are assigned to g_1 which simply represents necessary external buses. Next, the largest of the elements of the terminal capacity matrix $(t_{i,j})$ are identifed. Vertices i and j are assigned to a group g_2. Any other vertices, k, where $t_{k,\ell}$ is the same as $t_{i,j}$ where $\ell \varepsilon g_2$, are also assigned to group g_2. Next, for each $t_{m,n} = t_{i,j}$, where neither m nor n are assigned to any group, m and n are assigned to a new group. Then, for any other vertices, k, where $t_{k,\ell} = t_{i,j}$, for $\ell \varepsilon g_i$ and $k \not\varepsilon g_j$ for any j, vertex k is added to group g_i. The vertices of a group are placed next to each other in the terminal capacity matrix. Any vertex k with the next highest values of $t_{k,\ell}$ where ℓ is assigned to a group are placed close to the group in the reordered terminal capacity matrix.

Equation B.3 is the reordered terminal capacity matrix for the example used here.

Added in this equation are weights for each vertex representing the relative amount of chip area requried by each vertex. In this example node 3 is given weight of 1-1/2 since it represents the instruction decode circuitry as well as the IR register. The vertex weights are indicated in parentheses above the column vertex label. These weights (w_i = weight of vertex i) are used to determine whether the amount of area required for

$$T_2 = \begin{array}{c|cc|cccccc|ccc|}
 & (0) & (0) & (0) & (1\frac{1}{2}) & (1) & (1) & (1) & (1) & (0) & (1) & (0) \\
 & 1 & 8 & 2 & 3 & 4 & 5 & 9 & 11 & 6 & 7 & 10 \\
1 & d & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
8 & 1 & d & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline
2 & 1 & 1 & d & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
3 & 1 & 1 & 1 & d & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
4 & 1 & 1 & 1 & 1 & d & 2 & 2 & 2 & 2 & 2 & 2 \\
5 & 1 & 1 & 1 & 1 & 2 & d & 2 & 2 & 2 & 2 & 2 \\
9 & 1 & 1 & 1 & 1 & 2 & 2 & d & 2 & 2 & 2 & 2 \\
11 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & d & 2 & 2 & 2 \\ \hline
6 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & d & 3 & 3 \\
7 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & d & 3 \\
10 & 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & d
\end{array} \qquad (B.3)$$

any particular group is less than the maximum (W_{MAX}) able to be implemented with one chip. A maximum connection limit denoted C_{MAX}, must not be exceeded by each group which fits onto one chip. These limits will be used to find the final partition using the following procedure. The values of these limits for the example will be $W_{MAX} = 3$, $C_{MAX} = 3$.

Step 3 involves forming the connection matrix for the reordered terminal capacity matrix. Step 2 found groups with many connection paths, but actual connections is the main criterion for the final partition and the connection matrix must be used for this. The reordered connection matrix for the example used here is in Equation B.4.

$$C_2 = \begin{array}{c|cc|cccccc|ccc|}
 & (0) & (0) & (0) & (1\frac{1}{2}) & (1) & (1) & (1) & (1) & (0) & (1) & (0) \\
 & 1 & 8 & 2 & 3 & 4 & 5 & 9 & 11 & 6 & 7 & 10 \\
1 & d & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
8 & 0 & d & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ \hline
2 & 1 & 0 & d & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 1 & d & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & d & 2 & 0 & 0 & 1 & 1 & 0 \\
5 & 0 & 0 & 0 & 0 & 2 & d & 0 & 0 & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & d & 0 & 0 & 1 & 1 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & d & 1 & 0 & 1 \\ \hline
6 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & d & 1 & 0 \\
7 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & d & 1 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & d
\end{array} \qquad (B.4)$$

The two groups formed in step two are $g_1 = (1,8)$ and $g_2 = (6,7,10)$. Each of the remaining vertices will be assigned to a group of one vertex. The groups assigned to each of the remaining vertices are as follows:

$g_3 = (2)$

$g_4 = (3)$

$g_5 = (4)$

$g_6 = (5)$

$g_7 = (9)$

$g_8 = (11)$

Merges of groups are now found to form final partitions, denoted P_i, which reduce the connections as much as possible.

Step 4 is the first merging step. First some definitions are given that will be used in the merging procedure.

A target group is a group with $W_i > 0$ which other groups are merged with to reduce the number of connections or to fill the merged group chip area. The other group is called the merged group.

A partitioned unit, denoted P_i, is a group that is finished being merged.

$R_{i,j}$ denotes the reduction of connections obtained by merging a group g_i with the target group, g_j.

$M_{i,j}$ represents a merging index which indicates a gain obtained by the merge which is equal to $R_{i,j}$ divided by the total weight of all vertices of the merged group, g_i.

The first part of Step 4 assigns any completed group as a partitioned

In the example g_1, consisting of the two external vertices, cannot be merged with any groups and is complete. Therefore group g, is assigned to P_i.

Merges of groups are now found. The group with the largest number of outside connections and $W_j > 0$ is chosen as the target group. The number of outside connections is equal to the sum of all $t_{i,j}$ where j is a vertex in the group and i is a vertex not in the group. In the example, group g_2 is the first target group.

Then the group which does not add any external connections to the target group and has the largest index $M_{i,j}$ with the target group is merged with it. This step is repeated for all groups as target groups. The order in which the target groups are chosen does not matter since if a group has connections only to one group when it is merged, there are no other groups with which it could be merged in this step. This merge is only carried out if W_{MAX} is not exceeded by performing the merge. This step never adds any connections to any group. It only reduces them.

In the example, groups g_7 and g_8 are merged with group g_2 which forms a group with 3 external connections and a weight of 3. This group is completed since W_{MAX} has been reached and no further merges are possible. It is now denoted as P_2. Group g_6 is merged with group g_5, which forms a group g_5 with weight 2 and $C_5 = 3$. The only groups left are $g_5 = (4,5)$, $g_3 = (2)$ and $g_4 = (3)$. No further merges of the type described in this step are possible.

Step 5 consists of finding merges which may add new connections to the group, but reduce the net total of connections to the group. Each group g_i, has a set of connections with the target group, g_j, and a set of connections g_i has w[illegible]h the target group is greater than the number g_i has with outside groups, then the total reduction of connections is the difference between these two sets of connections, which is equal to $R_{i,j}$. If the value of R_{ij} is positive for a merged group with a target group, then there are no other target groups with which this group will have a positive value of R_{ij}. This is true since if there were a second target group, g_{kg} for which R_{ik} is positive, then the merged group would have more connections with this second target group g_k than with all other groups. In this case R_{ij} must be negative. Therefore, there can be only one target group with which a merged group could have a positive value of R_{ij}.

The group with the largest number of connections and $W_j > 0$ is chosen as the first target group and merged groups are found which have a positive value of $R_{i,j}$. Of these groups, the group with the largest value of $M_{i,j}$ with this target group is merged first, if W_{MAX} is not exceeded. If W_{MAX} is exceeded the group with the next highest value M_{kj} is tried. This is repeated until either the target group is filled or no further merged groups are left. After trying all possibilities of merged groups, the next target group is chosen by taking the group with the next largest number of connections and $W_j > 0$.

In the example, after Step 4 the connection matrix is as shown in Equation B.5. The only groups left are g_3, g_4 and g_5. Both groups g_3 and g_5 have 3 external connections. Group g_5 has no groups which will reduce its number of external connections. Group g_4 reduces the number of

$$C_2 = \begin{array}{c|cc|c|c|cc|ccccc}
 & P_1 & & g_3 & g_4 & g_5 & & P_2 & & & & \\
 & (0) & (0) & (0) & (1\frac{1}{2}) & (1) & (1) & (1) & (1) & (0) & (1) & (0) \\
 & 1 & 8 & 2 & 3 & 4 & 5 & 9 & 11 & 6 & 7 & 10 \\
\hline
1 & d & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
8 & 0 & d & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
\hline
2 & 1 & 0 & d & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 1 & d & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
4 & 0 & 0 & 1 & 0 & d & 2 & 0 & 0 & 1 & 1 & 0 \\
5 & 0 & 0 & 0 & 0 & 2 & d & 0 & 0 & 0 & 0 & 0 \\
\hline
9 & 0 & 0 & 0 & 0 & 0 & 0 & d & 0 & 0 & 1 & 1 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & d & 1 & 0 & 1 \\
6 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & d & 1 & 0 \\
7 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & d & 1 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & d
\end{array} \qquad (B.5)$$

connections to group g_3 but g_3 has weight 0 and cannot be used as a target group. Therefore g_3 and g_4 cannot be merged. Considering the relative weights, group g_4 has a weight of $1\frac{1}{2}$ and 1 connection, group g_5 has a weight of 2 and 3 connections. These two groups cannot be merged since W_{MAX} would be exceeded and therefore g_4 is assigned to P_3 and g_5 to P_4. Group g_3 has weight zero and does not have to be assigned. The partition for this example is complete. The steps for the completion of the partition for the general case follow.

Step 6 merges any groups g_i with a target group g_j, where R_{ij} is 0. The group g_i which has the smallest weight is merged first if W_{MAX} is not exceeded. This is repeated until the group is filled or no further groups with zero value of $R_{i,j}$ are left. If W_{MAX} is exceeded by a merge then no further groups are tried in this step. This step is repeated for the rest of the groups as target groups.

Step 7 merges groups with weight > 0 which will increase the number of connections to the target group, but with the greatest values of $M_{i,j}$.

A target group, g_j, is chosen as the group with the greatest number of connections. Then groups, g_i, with $W > 0$ are examined and the one with the highest value of $M_{i,j}$ is merged first provided W_{MAX} and C_{MAX} are not exceeded. If W_{MAX} is exceeded, the group with the next highest value of $M_{i,j}$ is tried. If the merge is successful, the merged group is returned to the set of groups and a new target group chosen.

This step is repeated for each target group with new target groups chosen in decreasing order of the number of connections they have with other groups. If at any time C_{MAX} or W_{MAX} are met, the group is assigned to a partition unit and removed from the set of groups.

Step 8 has only groups which cannot be merged with another to form one chip. If C_{MAX} is not exceeded by a group, then this group is assigned to a partition unit. If C_{MAX} is exceeded the group must be implemented with a bit slice partition and Step 7 can be repeated with C_{MAX} nearly doubled (to allow some pins for duplicated control) and W_{MAX} doubled before bit slicing.

The final partition found for the example is shown in Table B.1.

Table B.1. Final Register Partition

P_i	Vertices	W_i	C_i
P_1	1,8	0	2
P_2	6,7,9,10,11	3	3
P_3	3	$1\frac{1}{2}$	1
P_4	4,5	2	3

VITA

William Joseph Kaminsky, Jr., was born in East Chicago, Indiana on September 18, 1951. He received the B.S. degree in Engineering from Purdue University C.C. in 1973 where he was selected as the Outstanding Student in the School of Engineering in 1973. He received the M.S. degree in Engineering from the University of Illinois in 1974. He held engineering positions with the Northern Indiana Public Service Co. from 1970-1973, Motorola, Inc. in 1974, and Lawrence Livermore Laboratory in 1975. At the University of Illinois he was employed as a Research Assistant in the Radio Location Laboratory from 1973 to 1974, a Teaching Assistant in the Department of Electrical Engineering from 1974 to 1975 and a Research Assistant at the Coordinated Science Laboratory from 1976 to 1977.